SCRIPT GUIDE

# QmodemPro

FOR WINDOWS

# SLIQ

## *Script Language Interface for QmodemPro*

**Windows Release**

(12/1/94 printing)

P.O. Box 2264

Bakersfield, CA 93303

Tech. Support (805) 873-2550 — Office & Sales (805) 873-2500

Fax (805) 873-2599 — BBS (805) 873-2400

Orders only (800) 999-9619

The information in this document is subject to change without notice and should not be construed as a commitment on the part of Mustang Software, Inc.

*Qmodem, QmodemPro, QmodemPro for Windows*, MSI and the MSI Horse Logo are trademarks of Mustang Software, Inc. Other products and corporate names may be trademarks or registered trademarks of other companies, and are used in this document for illustrative purposes only, and to the owners benefit, without an intent to infringe. The names of companies, persons and products used in this documentation are fictitious unless otherwise noted, and are included solely to document the program.

Copyright © 1993, 1994 Mustang Software, Inc. All rights reserved. No part of this document may be reproduced, transmitted or transcribed in any form or by any means, electronic or mechanical, including photocopying or recording, without the written permission of Mustang Software, Inc.

# Contents

**SLIQ** ..... **1**
Satisfaction Guarantee ..... 4
About This Guide ..... 5

**1 - Introduction** ..... **7**
Getting Started ..... 9
A Technical Overview of SLIQ ..... 17

**2 - Script Commands** ..... **31**
Script Command Reference ..... 33
Using DLL Functions ..... 253

**3 - Script Examples** ..... **255**
Script Examples ..... 257

**4 - Debugging** ..... **263**
Debugging A Script ..... 265
Compiler Errors ..... 268
Runtime Error Messages ..... 280

**Index** ..... **283**

# Satisfaction Guarantee

Mustang Software wants you to be satisfied with your purchase of *QmodemPro for Windows*. If for any reason you are not satisfied and wish to return *QmodemPro for Windows*, you may do so within the first 30 days after purchase. Simply return the entire contents of the package, including the manual, disks, and registration card, to your dealer. Proof of purchase date is required. Returns direct to MSI can only be arranged if the product was purchased direct from MSI. Direct returns must be pre-approved and a return authorization number must be obtained in advance.

# About This Guide

This script reference guide is designed to help you write scripts using SLIQ (*Script Language Interface for QmodemPro*). It is not designed to teach you programming or the BASIC programming language.

Chapter 1 provides an overview of SLIQ.

Chapter 2 contains a complete list of SLIQ commands and is designed to be used as a reference. This chapter also contains complete information on using dialog boxes and external DLL functions.

Chapter 3 helps you put the information from the previous chapters to work by providing specific examples of script source code.

Chapter 4 documents error codes and the use of the script debugger.

## Getting additional information

In addition to the information in this guide there are numerous sources for help and training on the BASIC programming language, the underlying language of SLIQ. Your favorite bookstore or computer user group can be an invaluable source of help. Look for a tutor on the BASIC implementation used by Quick Basic from Microsoft and you will be on the right track.

# 1 - Introduction

*Education is what survives after what has been learnt has been forgotten.*

**B.F. Skinner**

# In this chapter

Getting Started ..... 9
- What is a script? ..... 9
- What can I do with a script? ..... 9
- Design elements ..... 10
- How are scripts created? ..... 12
- Compiling a script ..... 13
- Debugging a script ..... 14
- Running a script ..... 14
- Stopping a script ..... 16

A Technical Overview of SLIQ ..... 17
- General Program Information ..... 17
- Constants ..... 18
- Types ..... 19
- Variables ..... 21
- Expressions ..... 22
- Statements ..... 23
- Subroutines and Functions ..... 28

# Getting Started

## What is a script?

A script program is a set of instructions that you give *QmodemPro for Windows* that tells it how to perform a particular task. Its structure and syntax is similar in many ways to the popular Basic programming language, but you don't need to be a programmer to make use of SLIQ scripts.

Depending on the complexity of its construction, a SLIQ script can be thought of as a command shortcut, a macro or a computer program. Scripts are an ideal way to automate repetitive tasks, but can also be used to create custom options for the flow of your telecommunications activity. A SLIQ script can make use of *QmodemPro for Windows* program functions as well as internal functions within Windows itself.

SLIQ scripts are composed of lines of text that represent the activities to be performed. It's a compiled script language, meaning that the text representation of the script is converted to a binary file when it is complete. Compiling scripts not only allows them to execute faster, but lets you distribute executable copies of your scripts without the underlying text source code.

For the large majority of *QmodemPro for Windows* users, script usage will be limited to grouping a sequence of tasks into a single command, usually by using the QuickLearn feature. QuickLearn, as you'll see later, provides an interface to SLIQ that is as easy to use as a tape recorder; you just turn it on and it records your task for playback at a later time.

## What can I do with a script?

SLIQ provides the means of performing almost any telecommunications activity desired.

## Automated logon

While the steps to connect to various on-line services, host computers and BBS systems are very straightforward in *QmodemPro for Windows*, each host has its own procedures for getting your name, password and other information. Automation of this process is probably the most common application for a script program. It is also the one that lends itself best to the QuickLearn feature since it is repetitive in nature.

## File transfers

Automating the process of downloading files from a remote system is another application where scripts are handy. Although the QuickLearn feature can also be used for this type of activity, it may require modification to control the desired download directories or other download parameters.

## Fully automated sessions

Combining the two previous examples you can see that a more complex script is capable of logging on to a remote system, downloading files and perhaps additional activity such as retrieving new messages. Any activity that can be performed manually with *QmodemPro for Windows* can be automated using SLIQ.

## Unattended *QmodemPro for Windows* operations

Perhaps the best example of an unattended script is the *QmodemPro for Windows* host mode. The functionality of host mode is discussed in the User Guide, but the actual operation of host mode is driven by a series of pre-written scripts that are included with *QmodemPro for Windows*. These scripts and several others will be used as examples as we explore the power of SLIQ.

# Design elements

As indicated earlier, SLIQ is patterned after the BASIC programming language. This decision is based on the desire to incorporate a number of

advanced programming structures and concepts, while appealing to the largest audience. If you are familiar with BASIC you will be accustomed to many aspects of SLIQ. There will be some new commands specific to *QmodemPro for Windows* but you should feel very comfortable with writing and editing scripts.

If you aren't yet familiar with BASIC we encourage you to give SLIQ a try. BASIC was designed to be fairly easy to learn, and with the help of a book or two on BASIC you may be well on the way to writing very productive scripts. There are many excellent reference and training books on the BASIC language at your local bookstore or user group library. It is beyond the scope of this manual to actually teach the BASIC language, and we admit that we make little effort to do so in this documentation.

SLIQ scripts must be compiled to an executable format before they can be used. Compiled scripts are advantageous because they execute faster than interpreted scripts and allow the author to keep the script source code confidential. If an attempt is made to execute a script before it has been compiled, the compiler is automatically invoked by the program. Once compiled, either manually or automatically, a script is executed immediately when requested.

Scripts are stored in the directory specified for script files which is selected with the menu choice **Options/Files/Path Definitions**. SLIQ files use standard DOS filenames with two different extensions. The source script files created with your editor uses the extension **.SCR** and when they are compiled into an executable form by SLIQ they are given the extension .SCC.

Script files are created and modified using any standard ASCII text editor. *QmodemPro for Windows* includes an internal editor that you access from the menu choices **Scripts/Edit**. The internal editor supports auto-indent and tabs for structuring your script source files in a readable format, however you are free to use an external editor or programming environment if desired.

## How are scripts created?

### The QuickLearn feature

The *QmodemPro for Windows* script language can be used for many applications without the need to learn anything about scripting or SLIQ. QuickLearn lets you to record keystrokes for playback at a later time and is a simple process.

The first way of starting a QuickLearn script is to revise a Phonebook entry and place the name of a new script in the Scripts field. The next time you dial this entry, *QmodemPro for Windows* will automatically record a script file.

The second way to create a QuickLearn script is to use the pull down menu choice **Scripts/QuickLearn** to begin recording a script.

When activated, *QmodemPro for Windows* displays the standard file selection dialog and request a name for the script to be created. All scripts should be saved in the directory designated for scripts which by default is C:\QMWIN\SCRIPTS. If your configuration is different you should save your scripts in the correct script directory.

Any valid DOS filename may be used and the extension must be **.SCR**. This extension is automatically added when script names are specified in the script filename dialog box.

Once a filename is selected you are returned to terminal mode and every keystroke is recorded until QuickLearn mode is terminated by again selecting the **Scripts/QuickLearn** menu choice. While recording is active the terminal window status line displays the name of the script preceded by the "greater than" symbol (>) indicating that recording is active. In addition, a check mark is placed in front of the QuickLearn menu option.

Scripts created using QuickLearn can be viewed and edited just as any other script. The menu choice **Scripts/Edit** can be used to take a look at the scripts created with SLIQ's QuickLearn feature.

While QuickLearn is an extremely convenient facility, it has its limitations. QuickLearn can record your interactions with a remote computer and replicate them exactly, but it cannot add decision-making logic to the scripts it creates, or account for unexpected or changing conditions. When you run out of QuickLearn power, it's time to write or edit your own scripts.

## Script programming

Writing a new script from the ground up is a necessity in some cases. Any standard ASCII text editor can be used to create or modify a SLIQ script, including the *QmodemPro for Windows* internal editor.

To create a new script select the menu choice **Scripts/Edit** to display the standard file selection dialog box. The default directory in the dialog will display files in your scripts directory as defined in **Options/Files/Path Definitions**, and the default extension will be **.SCR**. Create a new script file by entering the name of the script to be created. It is not necessary to add the script extension **.SCR** since it is added automatically when the **Script/Edit** menu command is used to create a script file. Note that the alternate method of invoking the editor, using the menu choice **Files/Edit a File** does *not* automatically add the extension **.SCR**.

To edit a SLIQ script use the same procedure as creating a new script, but select the desired script from the dialog box.

## Compiling a script

SLIQ is a compiled language. This means that the text of every **.SCR** text file is converted to executable code before being used by *QmodemPro for Windows*. Script source files (those ending in **.SCR**) are compiled either automatically or manually.

After creating a SLIQ script file you can manually perform a compile by selecting the **Scripts/Compile** menu choice. The compile process will read your script, check each line for proper syntax and create an executable **.SCC** file with the same name as the source file. If any errors are encountered the compile process will stop with a compile error

message and invoke the editor. The editor cursor will be located as near the error as possible, enabling you to make the necessary changes. A complete list of compiler error messages can be found in the sections **Compiler Errors and Script Debugging**, later in this manual.

If a script has been created or modified but not yet compiled when a request to execute is received, the compile process is automatically invoked. The compile process may take a few seconds to several minutes depending on the size and complexity of the script.

## Debugging a script

Regardless of the simplicity of a script or the capabilities of the programmer, scripts are seldom written perfectly on the first attempt. Many times an error is simply a typographical mistake or improper syntax for a command. In these cases the errors are caught during the compile cycle and the editor cursor is placed at the approximate location of the error in the script source. This type of error is usually relatively easy to correct.

The harder type of error to locate and correct involves a script that follows correct syntax and compile guidelines, but fails to produce the desired results. In this case the script debugger is an invaluable tool. The debugger allows you to execute a script line-by-line while viewing the results. It can skip sections of code and even display the changing value of variables as each line is executed.

The operation of the debugger is discussed in its own section after the SLIQ language documentation. See the **Compiler Errors** section under **Debugging** for details.

## Running a script

Scripts can be executed in several different ways:

1. Scripts can be started with the menu command **Scripts/Execute**. When this option is used, you can select either the **.SCR** or **.SCC** filename for the desired script if both exist. If only the **.SCR** source filename exists, the script will be compiled to a **.SCC** file before execution.

2. Scripts can be executed by clicking on the toolbar button labeled **Run Script**. This action displays the same dialog box that is called when the **Scripts/Execute** command is selected.

3. Scripts can be started automatically every time a Phonebook entry is dialed. A script name is associated with a specific dialing directory entry in the *Script* field for entries in the *QmodemPro for Windows* Phonebook. When connection is made, the script is executed automatically. These types of scripts are often referred to as *linked scripts* because they're linked to phonebook entries. The same script can be tied to multiple Phonebook entries, and can be used to take control immediately after a connection to automate repetitive logon processes. Linked scripts that automate logon activities are also called *logon scripts*.

4. Scripts can be started by clicking a macro key on the macro bar at the bottom of the screen that has been assigned to a script name using the @SCRIPT macro command. Once started, macro-linked scripts operate the same as a script executed from the pull-down menu.

5. Scripts can be executed immediately when *QmodemPro for Windows* is started. The command line syntax for launching *QmodemPro for Windows* with a script is:

   ```
   QMWIN FILENAME.SCC
   ```

   or

   ```
   QMWIN FILENAME.SCR
   ```

   which will cause a compile if needed.

   Once initialization is complete, *QmodemPro for Windows* will load and execute the script specified on the command line.

6. Scripts can be executed automatically by creating an association in Windows File Manager between the SLIQ extensions (.SCR. and .SCC) and the *QmodemPro for Windows* program QMWIN.EXE. This association allows you to double-click on the script file name

within File Manager to launch *QmodemPro for Windows* with the selected script as a command line argument.

Regardless of how a script is invoked, it operates in the same manner. The Script icon on the toolbar is displayed in a depressed position during script execution and the terminal windows status line displays the name of the script being run.

## Stopping a script

You can stop a script at any time in several ways. The easiest is to click on the **Script** icon on the toolbar. A confirmation prompt dialog box is displayed before execution is terminated.

You can also toggle the menu selection **Scripts/Execute** to stop a script. This menu item displays a check mark during execution, which is removed after confirming that you want to stop the script.

# A Technical Overview of SLIQ

## General Program Information

Every variation of BASIC is slightly different, and SLIQ is no exception. This section provides information about the various items that make up SLIQ and how they operate. It only deals with the underlying operational characteristics of the language, not the command set. The entire set of SLIQ commands are discussed in the chapter that follows.

### Tokens

Tokens are the smallest meaningful units of text in a *QmodemPro for Windows* script program. There are five kinds of tokens: special symbols, numbers, keywords, identifiers, and string constants.

### Special Symbols

Besides letters, numbers, and spaces, SLIQ accepts the following special characters and pairs of characters:

```
+ - * / = < > . , ( ) : ; ' " <= >= <>
```

### Numbers

SLIQ accepts numbers in ordinary decimal format. Hexadecimal numbers are represented using 0x or &H as a prefix to the number. Engineering notation (e or E, followed by an exponent) is read as "times ten to the power of" in real numbers. For example, 7E-2 means $7 \times 10^{-2}$.

### Keywords

Keywords appear in lower case `typewriter font` throughout this section. The following is a complete list of SLIQ keywords:

```
access alias all and append as binary call caption case catch close
const declare dial dialog dialogbox dim do else elseif end eqv exit
flush font for function get gosub goto if imp include input is len let
lib lock loop mid mod name next not open or output print put random
read receive rem return seek select send shared static step sub then
to type unlock until wend when while write xor
```

These keywords cannot be used as identifiers. Since SLIQ is not case sensitive, these keywords may appear in any combination of upper or lower case in your scripts.

## Identifiers

Identifiers denote constants, types, variables, subroutines, functions, and fields in user defined types. An identifier can be of any length, but shorter identifiers are usually easier to manage. An identifier must begin with a letter and can not contain spaces. Letters, digits, and underscore characters are allowed after the first character. Like keywords, identifiers are not case sensitive.

## String Constants

A character string is a sequence of zero or more characters from the extended ASCII character set, written on one line in the program and enclosed by double quote characters ("). Case is significant in string constants; "Hello" is a different string from "HELLO".

## Comments

Comments consists of text the compiler should ignore, but are useful for annotating your SLIQ programs. Comments are preceded by either the `rem` statement or the ' character (single apostrophe).

Note that `rem` is treated as a regular statement, so if you want to place a comment at the end of a line using `rem` , you must precede the `rem` by a colon to separate it from the previous statement. The single apostrophe can be used without the need for a separating colon:

```
print "Hello world!"    'comment OK
print "Hello world!"    :rem comment OK
```

Note that the first needs no colon while the second does.

## Constants

A constant is an identifier that represents a value that can't change. A constant is declared in a SLIQ program like this:

```
const Tries = 10
```

This declaration causes the value 10 to be substituted wherever the identifier `Tries` appears in the program. You can also define constants in terms of other constants using expressions, like this:

```
const LookingBad = Tries - 3
```

This will cause the value *7* to be used wherever `LookingBad` appears in the program.

## Types

SLIQ is a typed language, which means that each variable or expression has an associated type. This type defines the kind and size of values that the variable can hold, as well as the operations that can be performed on that variable. The following identifiers represent the predefined types in the language:

```
byte
integer
long
real
string
```

### Simple types

There are five simple types whose characteristics are shown in the following table:

| Type | Range | Format |
|---|---|---|
| byte | 0 to 255 | 8 bit unsigned |
| integer | -32768 to 32767 | 16 bit signed |
| long | -2147483648 to 214783647 | 32 bit signed |
| real | 1.5e-45 to 3.4e38 | 32 bit IEEE |
| string | 0 to 32767 characters in length | variable length |

Strings can also be declared to be of fixed length, which means that they always hold a specific number of characters (even if those characters are just spaces). Here is an example of a declaration of a string of length 30:

```
dim name as string*30
```

In this case `name` will always hold exactly 30 characters. Fixed length strings are particularly useful in user defined types (see below) because variable length strings are not permitted in a user defined type declaration.

## Arrays

Arrays are sequences of values, all of which have the same type. For example, the following code declares an array of 10 integers:

```
dim a(10) as integer
```

Actually, there are 11 integers declared here; they are referenced as a(0), a(1), through a(10). The type of elements in an array can be of any type except another array.

## User Defined Types

A user defined type is a type that holds more than one value. Unlike an array, the values that a user defined type holds can be of different types. For example, here is a user defined type that can be used to hold a date value:

```
type date
  year as integer
  month as integer
  day as integer
end type
```

In this example all the fields are of the same type. Here is another example that could be used to hold a person's information:

```
type person
  name as string*50
  phone as string*20
  birthdate as date
```

```
end type
```

Note that the `date` type declared above can be used inside another type declaration. User defined types can be nested in this way as many levels deep as necessary.

## Dialog Box Types

SLIQ supports Windows dialog boxes using a mechanism similar to user defined types. For information on declaring and using dialog boxes, see the separate section on Dialog Boxes at the end of chapter 2.

## Variables

A variable is an identifier that represents a value that can change. A variable is declared in a SLIQ program like this:

```
dim count as integer
```

The word "count" can then be used in any expression where an integer is accepted. To declare an array of values, place the number of values you want to declare in parentheses after the variable identifier. For example,

```
dim friends(10) as string
```

This causes friends(0) through friends(10) to be declared. Declaring a user defined type works the same way:

```
dim bob as person
bob.name = "Bob Smith"
bob.phone = "555-1212"
bob.birthdate.year = 1963
bob.birthdate.month = 4
bob.birthdate.day = 17
```

The above statements also assign values to each of `bob`'s fields. As you can see, we have built upon our previous examples to create a personal record.

## Expressions

Expressions are made up of operators and operands. Most operators are binary and take two operands. Two operators are unary and only take one operand. Binary operators use the usual algebraic form (for example, A + B). A unary operator always precedes its operand (for example, -B). There are two unary operators, `not` and `-`. `Not` is a bitwise logical not of its operand, and - takes the negative of its operand. The following is a list of the binary operators in SLIQ, in order of precedence from highest to lowest:

| Operator | Notes |
|---|---|
| `^` | exponentiation |
| `* / \ mod` | multiplication, division, integer division, modulus |
| `+ -` | addition, subtraction |
| `= <> < <= > >=` | equality and inequality operators |
| `and` | bitwise logical and |
| `or` | bitwise logical or |
| `xor eqv` | exclusive or, equivalence ((x eqv y) = not (x xor y)) |
| `imp` | implication, ((x imp y) = ((not a) or b)) |

Parentheses are used to modify the above precedence order. Expressions within parentheses are always evaluated starting with the innermost set of parentheses. For example,

| Expression | Notes |
|---|---|
| 3 + 4 * 5 = 23 | (remember, multiplication takes precedence over addition) |
| (3 + 4) * 5 = 35 | |

Functions are called by naming the function to call, perhaps followed by a parameter list in parentheses after the function name. For example, if s is a string,

```
len(s)
```

is an integer type expression equal to the length of the string s.

## Statements

Statements are commands that tell SLIQ what to do. Statements can be broken into two major classes — those that control the execution of other statements and those that don't.

Simple statements stand by themselves and don't control the execution of other statements. A simple statement is our first example:

```
print "Hello world!"
```

Compound statements directly control the execution of other statements. Each type of compound statement is described in the following sections:

### Single line if statement

The simplest sort of compound statement is the single-line if statement:

```
if x = 5 then print "x is five"
```

This causes the print statement to be executed only if x currently has the value 5. If x does not have the value 5, then the print statement is skipped.

### Multiple line if statement

If statements can also span multiple lines, as in this example:

```
if x = 5 then
  print "x is five"
  x = x + 1
  print "x is now six"
end if
```

The style of indenting the lines that are controlled by the if statement is not required, but it makes the program easier to read. You will see this style in each of the following compound statement examples. If statements can also have an else directive:

```
if x >= 1 and x <= 10 then
  print "x is between 1 and 10"
else
  print "x is less than 1 or greater than 10"
end if
```

Finally, if statements can have an "elseif" directive, which helps make certain constructs easier to write (see the select case statement below for another approach):

```
if x >= 1 and x <= 10 then
  print "x is between 1 and 10"
elseif x > 10 then
  print "x is greater than 10"
else
  print "x is less than 1"
end if
```

## Loop statement

The loop compound statement comes in four flavors which differ in when and how the condition to exit the loop is tested. In general, the loop statement will repetitively execute a group of statements until a particular condition is met. For example,

```
x = 1
do while x <= 5
  print "x is "; x
  x = x + 1
loop
```

This is what is called a top-tested loop — the condition is tested at the top of the loop before any of the statements inside the loop are executed. In this case the loop will be executed five times — once for each value of x from 1 to 5. The other kind of top-tested loop is shown below:

```
x = 1
do until x > 5
  print "x is "; x
  x = x + 1
loop
```

The above example functions exactly the same as the first, the difference is the kind of test that is performed to see whether the loop should exit.

Bottom-tested loops work in a similar way, except that the test to see whether the loop should exit occurs after all the statements in the loop. The statements in a bottom-tested loop are always executed at least once (in a top-tested loop the statements may not be executed at all).

```
x = 1
do
  print "x is "; x
  x = x + 1
loop while x <= 5
```

or

```
x = 1
do
  print "x is "; x
  x = x + 1
loop until x > 5
```

An alternate form of the top-tested while loop is shown below:

```
x = 1
while x <= 5
  print "x is "; x
  x = x + 1
wend
```

## Select case statement

The select case statement is a convenient way of testing a number of similar conditions. For example, the following statement tests the value of a variable x:

```
select case x
  case 1
    print "x is 1"
  case 2 to 5
    print "x is between 2 and 5"
  case is > 10
    print "x is greater than 10"
  case else
    print "x is less than 1 or between 6 and 10"
end case
```

There are several different types of cases that can be tested: single value, range of values, and a relation. An example of each is shown in the above code example. A single case value matches if the case variable is exactly the same as the value given. A range case checks the case variable against the given values to see whether it is within the given range including the endpoints. In the above example, x would be tested as "`x >= 2 and x <= 5`". A relation case is denoted by the keyword `is`, followed by a binary operator, followed by a value. The keyword `is` is replaced by the case variable (x in the above example) and the relation is checked. The `case else` clause is executed if none of the above conditions match.

## For statement

The for loop is a convenient way of performing a task a number of times, particularly when the number of iterations is known beforehand. For example:

```
for i = 1 to 10
  print "i is "; i
next i
```

In a for loop, the control variable (i in the above case) can be either increasing or decreasing. You may also specify a step value if you want the control variable to change by a value other than one. Here is a loop that prints the odd numbers from 9 to 1 backwards:

```
for i = 9 to 1 step -2
  print i
next
```

In the `next` statement that ends a for loop, it is not necessary to name the control variable as in the above example. If the control variable is specified it must match the most recent for loop. For example, the following is illegal:

```
for i = 1 to 5
  for j = 1 to 10
    print i, j
  next i          ' error, i not last loop
next j            ' j invalid here as well
```

## When statement

The when statement is a structured way of executing statements at the time a particular event occurs. There are a number of different events which can trigger a when statement:

- a time duration has elapsed
- a certain time of day occurs
- a sequence of characters is received from the communications port

The structure of a when statement is very much like the structure of an if statement. For example,

```
when match "press enter" do
  send
end when
```

This will cause the send statement to be executed whenever the characters "press enter" are received from the communications port. A when statement remains active and can be triggered again until it is explicitly cleared with a `clear when` statement.

## Subroutines and Functions

Subroutines and functions can be declared to help perform repetitive tasks. Here is a simple subroutine declaration:

```
sub test
  print "in sub test"
end sub
```

This subroutine can be called in one of two ways:

```
test
call test
```

The call keyword is optional here. Arguments can also be passed to functions, for example:

```
sub test(a as integer, b as integer)
  print "the sum of "; a; " and "; b; " is "; a+b
end sub

call test(1, 2)
call test(6, 11)
test 3, 4 ' notice that parentheses around the
          ' arguments are only allowed with the call keyword
```

The output of this program is:

```
the sum of 1 and 2 is 3
the sum of 6 and 11 is 17
the sum of 3 and 4 is 7
```

Parameters are normally passed by reference, which means that if a subroutine changes the value of a variable, the change will be reflected in

the variable which was used in the original parameter list. The exception to this is if an expression is passed to a procedure. For example,

```
sub bump(a as integer)
  a = a + 1
end sub

dim x as integer
x = 3
print x
bump x
print x
bump x+1
print x
bump (x)
print x
```

The output of this program is 3, 4, 4, 4. In the first call to bump, x is actually incremented because it is directly passed to the variable a in the bump subroutine. In the second call to bump, x is not incremented because it is not directly passed to the subroutine — x+1 is passed instead. In the third example, x is also not directly passed because it is enclosed in parentheses.

Functions are similar to procedures, except that they always return a value and this return value can be used in expressions. For example,

```
function f(a as integer, b as integer)
  f = a * b
end function
```

This declares a function which multiplies its arguments and returns their product. It can be used like this:

```
print f(5, 6)
x = y + f(z, w)
```

The first line prints the value 30, and the second line will first multiply z and w, then add that to y, and assign the result to x.

# 2 - Script Commands

*The proper study of mankind is books.*

**Aldous Huxley**

# In this chapter

Script Command Reference ..... 33
Using DLL Functions..... 253

# Script Command Reference

The following pages contain a complete description of all available *QmodemPro for Windows* script commands. Script commands are presented in the following format:

## Script Command

A brief description of the command.

### Syntax

```
A description of the command's syntax.
```

### Remarks

A more detailed description of the command, including parameters and their meaning, and how the command interacts with the rest of the script and the system in general.

### See also

Related script commands.

### Example

```
Most examples will be a complete script program that you can type in
and run.
```

## + (Concatenation) Operator

Concatenates (combines) two or more strings into one string.

### Syntax

```
string1 + string2 [+ string3 ...]
```

### Remarks

This operator is used to combine strings into a single, longer string. For instance, you can concatenate DATE and TIME variables into a single string variable called NOW (see example).

No spaces or other characters are placed between the strings during concatenation.

### See also

LEFT, MID, RIGHT

### Example

This example combines the current date and time strings into one string called now, and prints it on the screen.

```
dim now as string
now = date + " " + time
print now
```

## ABS Function

Returns the absolute value of a numeric expression.

### Syntax

```
ABS(number)
```

### Remarks

*number* can be any valid numeric expression.

The absolute value of a number is the quantity of the number without regard for its sign (positive or negative). A negative number has the same absolute value as the corresponding positive number. In other words,

abs(x) = x  if x is positive

abs(x) = -x  if x is negative

### See also

SGN

### Example

This example shows how two different numbers can have the same absolute value.

```
dim i as integer, j as integer
i = 5
j = -5
if abs(i) = abs(j) then
  print "i and j have the same absolute value"
end if
```

## ACTIVATE Statement

Used to make *QmodemPro for Windows* the active application on the Windows desktop.

### Syntax

```
ACTIVATE
```

### Remarks

This command will also restore the *QmodemPro for Windows* window if it is currently minimized.

### See also

MINIMIZE, MAXIMIZE, MOVE, SIZE

### Example

This example minimizes the *QmodemPro for Windows* window, waits for the string "connect" from the modem, then reactivates the application.

```
minimize
waitfor "connect"
activate
```

## ADDENTRY Statement

The ADDENTRY statement is used to add a new dialing directory entry to the phonebook.

### Syntax

```
ADDENTRY phoneentry
```

### Remarks

The *phoneentry* variable is a variable of type PHONEENTRY, which has this declaration:

```
type phoneentry
  systemname as string*27
  number(1 to 5) as string*19
  userid as string*30
  password as string*25
  notefile as string*79
  scriptfile as string*79
  macrofile as string*79
  icondir as string*79
  emulation as integer
  protocol as integer
  faxonly as integer
  retries as integer
  device as string*20
  connects as long
  uploads as long
  downloads as long
  lastdate as long
  lasttime as long
  iconfile as string*79
  iconid as integer
end type
```

## See also

DIAL

## Example

This example shows how a new entry might be added to the phonebook.

```
dim entry as phoneentry
entry.systemname = "MSI HQ BBS"
entry.number(1) = "1-805-873-2400"
entry.scriptfile = "msihq.scr"
entry.emulation = ripscrip
entry.protocol = zmodem
addentry entry
```

## ADDLFTOCR Function

Used to get or set the current state of the "Add LF to CR" toggle.

### Syntax

```
ADDLFTOCR
```

or

```
ADDLFTOCR(onoff)
```

### Remarks

There are two forms to the ADDLFTOCR function. The first form takes no arguments and simply returns the current setting. The second form sets the tate of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means line feeds are added to incoming carriage returns. A zero value means no additional characters are added.

### See also

DUPLEX

### Example

This example shows how the ADDLFTOCR function can be used in a logon script to turn on the line feed control.

```
dim oldcrlf as integer
oldcrlf = addlftocr(on)
waitfor "UserID"
send "123456"
addlftocr oldcrlf
```

## AND Operator

AND performs a bitwise AND operation between its operands.

### Syntax

```
op1 AND op2
```

### Remarks

*op1* and *op2* are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the AND operator:

| x | y | x AND y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### See also

EQV, IF, IMP, NOT, OR, XOR

### Example

The first part of this example shows how the AND operator can be used as a bitwise logical operator to perform manipulations on integers. The second part of this example shows how the AND operator can be used to combine the results of two comparisons and supply the result to an IF statement.

```
dim i as integer
i = 129
if i and 127 = 1 then
  print "yes"
end if

dim j as integer
j = 4
if i = 129 and j = 4 then
  print "yes again"
end if
```

## ASC Function

Returns the ASCII value (from 0 to 255) of the first character in a string.

### Syntax

```
ASC(string)
```

### Remarks

*string* is any string expression. The return value of this function is the ASCII value of the first character in the string. If the string is empty this function returns zero.

The opposite of the ASC function is the CHR function. The CHR function returns the character corresponding to a particular ASCII code.

### See also

CHR, VAL

### Example

This example prints 104, which is the ASCII value of "h", the first character in the string "hello".

```
dim a as string
a = "hello"
print asc(a)
```

## ATN Function

Returns the arctangent of a numeric expression.

### Syntax

```
ATN(number)
```

### Remarks

The arctangent of *number* is the angle in radians whose tangent is equal to *number*. You can convert radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

### See also

COS, SIN, TAN

### Example

This example uses the atn function to compute the value of pi.

```
print 4*atn(1)
```

## AUTOANSWER Statement

Used to send the "Auto answer on" or "Auto answer off" string from the modem configuration.

### Syntax

```
AUTOANSWER onoff
```

### Remarks

When called with a nonzero argument (ON), this command sends the "Answer on" string from the modem configuration. When called with a zero argument (OFF), this command sends the "Answer off" string from the modem configuration.

### See also

DIAL, HANGUP, RESETMODEM

### Example

This example shows how the AUTOANSWER command might be used to answer an incoming phone call.

```
autoanswer on
loop
until carrier
```

## BEEP Statement

Causes *QmodemPro* to sound the regular Windows beep sound.

### Syntax

```
BEEP
```

### Remarks

If you have attached a .wav file to the "Default Beep" sound in Windows Control Panel, then this command will play the corresponding .wav file.

### See also

SOUND

### Example

This example simply sounds the Windows default beep.

```
beep
```

## BREAK Statement

Sends a break signal to the communications port.

### Syntax

```
BREAK
```

### Remarks

A break is a special signal sent to the communications port that lasts for a specific amount of time (the time that a break lasts is determined by the **Options/Device/Connection/Break length** setting).

### See also

HANGUP

### Example

This example sends a break signal, waits for the string "connection lost", then ends the script.

```
break
waitfor "connection lost"
end
```

## BYTE Type

Used to declare a variable that can handle byte sized numbers.

### Remarks

Variables of byte type can hold values that range from 0 to 255.

### See also

DIM, INTEGER, LONG, REAL, TYPE

### Example

This example declares a variable of type BYTE and assigns a value to it.

```
dim i as byte
i = 5
print i
```

## BYVAL Identifier

Indicate the parameter passing method for Subs and Functions.

### Syntax

```
Function Name(ByVal tStr As String) As String

Sub Name(ByVal tNum As Long)
```

### Remarks

The **ByVal** keyword allows you to define how a data type is to be handled within a subroutine at the time of designing the **Sub** or **Function**. The default method for handling parameters is to assign by reference which can be over-ridden by the use of parentheses when a call is made to the subroutine.

By using **ByVal**, it will force the compiler to create a working copy of the data type and leave the passed parameter untouched on return.

### See also

FUNCTION, SUB

## Example

```
Rem The difference in passing by value or by reference

Dim tStr As String
Function Extend(aStr As String, aLen As Word) As String
  If Len(aStr) < aLen Then
    aStr = aStr + String(aLen - Len(AsTr), ".")
  End If
  Extend = aStr
End Function

Function ExtendBV(ByVal aStr As String, aLen As Word) As String
  If Len(aStr) < aLen Then
    aStr = Pad(aStr, aLen)
  End If
  ExtendBV = aStr
End Function

tStr = "Testing this here String!"
Print "["; ExtendBV(tStr, 60); "]"
Print "["; tStr; "]" : Rem ByVal = no change
Print "["; Extend(tStr, 60); "]"
Print "["; tStr; "]" : Rem By reference = change
```

## CALL Statement

Executes a subroutine or function.

### Syntax

```
CALL name[( arg [, arg ...])]
or
name [arg[, arg ...]]
```

### Remarks

*name* is the name of the subroutine or function to execute. Subroutines and functions must be at least declared before you can call them (see the DECLARE statement for information on declaring subroutines and functions).

*arg* is an argument that is passed to the sub-program. Multiple arguments are separated with commas. When using the first syntax with the CALL keyword, parentheses are required around the argument list. When using the second syntax, omitting the CALL keyword, parentheses are not placed around the parameter list.

Arguments are normally passed to the subroutine or function in "reference" mode. This means that if the corresponding argument in the function is changed, the original copy will be changed too. This behavior may be changed by passing an expression to the function, like `a+1` or `(a)` (the parentheses around a serve only to create an expression and prevent a subroutine from changing its value). See the example below for an illustration of how this works.

### See also

DECLARE, FUNCTION, GOSUB, SUB

### Example

This example declares a subroutine called f which takes two arguments. The subroutine adds one to the first argument, and adds two to the second

argument. If you run this script, you will notice that the value of a in the main program is changed, while the value of b remains the same. This is because b is not directly passed to the subroutine, but a copy of b is. The subroutine changes the copy and does not affect the actual value of b.

```
declare sub f(x as integer, y as integer)
dim a as integer, b as integer
a = 4
b = 7
call f(a, (b))
print "a is "; a
print "b is "; b

sub f(x as integer, y as integer)
  x = x + 1
  y = y + 2
end sub
```

## CAPTURE Statement

Used to open or close a terminal capture file.

### Syntax

```
CAPTURE filename
CAPTURE ON
CAPTURE OFF
```

### Remarks

*filename* is the name of the file to which captured data will be appended.

**on** turns on the capture file specified in **Options/Files/File Definitions**.

**off** closes any currently open capture file.

If you specify a new capture file while another capture file is still open, the first capture file will be closed before the new one is opened.

If the capture file cannot be opened, the ERR_FILEOPEN error will be generated. This error can be caught with the CATCH statement.

### See also

CATCH, PRINTER

### Example

This example opens the capture file "test.cap", sends an Enter to the communications port, waits 10 seconds, then closes the capture file.

```
capture "test.cap"
send
delay 10
capture off
```

## CARRIER Function

Determines whether the modem is currently online and connected to another modem.

### Syntax

```
CARRIER
```

### Remarks

The CARRIER function returns TRUE if the modem currently reports that it is online and connected to another modem. If not, it returns FALSE. The return value of this function corresponds with the state of the "Online/Offline" indicator at the bottom of the terminal window.

### See Also

HANGUP

### Example

This example sends the string "bye" to the communications port, then waits until the modem reports that carrier is no longer active (that is, the remote modem has hung up).

```
send "bye"
while carrier do
wend
```

## CASE Statement

Introduces a new case in a SELECT CASE statement. Please see the description of the SELECT CASE statement for more information and examples.

## CATCH Statement

Used to catch runtime errors and perform error recovery actions.

### Syntax

```
CATCH errvalue [, errvalue ...]
or
CATCH ALL
```

### Remarks

The CATCH mechanism provides a convenient way of responding to errors that may occur while your script is running. The errvalue values must be one of the following:

#### ERR_ARRAYSUBSCRIPT

Caused by accessing an array using an invalid subscript index.

#### ERR_FILEOPEN

An error during a file open operation causes this error. The commands that can cause this exception are CAPTURE, OPEN, and SHELL.

#### ERR_FILERENAME

Caused by an error during a file rename operation. This could be caused by the original file not being found, or open by another application, or the destination filename is already in use. The NAME statement can cause this error.

#### ERR_FUNCTIONNOTFOUND

Caused by trying to call a function in a dynamic link library (DLL) where the named function does not exist.

## ERR_INVALIDFILENUMBER

Caused by using an invalid file number in any file operation, including OPEN, CLOSE, PRINT, INPUT, INKEY, LOF, and so on.

## ERR_LIBRARYNOTFOUND

Caused by trying to call a function in a dynamic link library (DLL) where the named DLL does not exist.

## ERR_MATH

Caused by trying to divide by zero, taking the logarithm of zero or a negative number, or by taking the square root of a negative number.

## ERR_PATH

Caused by an invalid drive or path name in one of the following commands: CHDIR, CHDRIVE, MKDIR, RMDIR.

## ERR_TIMEOUT

Caused by a timeout during one of the following commands: INPUT, RECEIVE, WAITFOR.

A CATCH statement may only be placed at the end of a user defined subroutine or function, or at the end of the main program body. During normal operation (the case where no runtime error occurs) the statements after the CATCH statement are skipped. If one of the above runtime errors occurs, the CATCH block for the currently executing function will be searched for a handler for the error. If there is no CATCH block or if there is no specific CATCH handler for the error that occurred, control will return to the function that called the current function. Its CATCH block (if any) will be searched for a handler, and so on up the call chain. If the error propagates all the way up to the main program body and there is no CATCH handler there for the error, then the script is automatically halted with an appropriate error message.

## See also

ERROR

## Example

This example defines a subroutine that opens a file called "test.dat", reads the first line of text from the file, and closes it. If there is an error opening the file, the subroutine prints an error message.

```
declare sub test
dim a as string
call test(a)
print a

sub test(s as string)
  open "test.dat" for input as #1
  input #1, s
  close #1
catch err_fileopen
  print "could not open test.dat"
end sub
```

## CHAIN Statement

Used to transfer control to another script.

### Syntax

```
CHAIN scriptname
```

### Remarks

This command terminates execution of the current script and starts execution of the named script. All currently open files are closed and variables are discarded.

The script named in this command must be the name of a compiled script with the proper .scc extension. Scripts are not automatically compiled by this command.

### See also

END, STOP

### Example

This example shows two files, a.scr and b.scr, and demonstrates how you can start execution of the second script from the first.

```
[in file a.scr]
print "in a.scr"
chain "b.scc"

[in file b.scr]
print "in b.scr"
```

## CHDIR Statement

Changes the current directory.

### Syntax

```
CHDIR directory
```

### Remarks

This command changes to the directory named in the *directory* argument.

As with the DOS CHDIR command, this does not change the current drive, even if it is specified in the argument. However, if you change the current directory for a drive that is not the current drive, the change will be remembered until you next change to the new drive using the CHDRIVE command.

### See also

CHDRIVE, CURDIR, CURDRIVE, MKDIR, RMDIR

### Example

This example shows various ways the CHDIR command can be used.

```
chdir "\"                'change to the root directory
chdir "c:\qmwin"         'change to C:\QMWIN
dim newdir as string
newdir = "c:\temp"       'assign specified directory to
                         'string variable NEWDIR
chdir newdir             'change to the new directory
```

## CHDRIVE Statement

Changes the current drive.

### Syntax

```
CHDRIVE driveletter
```

### Remarks

This command changes the current drive to *driveletter*. If *driveletter* is longer than one character, only the first character in the string is used.

### See also

CHDIR, CURDIR, CURDRIVE

### Example

This example takes advantage of the fact that only the first letter of the argument to CHDRIVE is used, and changes to the directory "c:\test" no matter which drive this script was started from.

```
dim a as string
a = "c:\test"
chdrive a
chdir a
```

## CHR Function

Returns the ASCII character corresponding to the specified ASCII code value in the range of 0 to 255.

### Syntax

```
CHR(number)
```

### Remarks

This function returns the ASCII character corresponding to *number*.

The opposite of the CHR function is the ASC function. The ASC function returns the ASCII character value of the first character in a string.

### See also

ASC, SPACE, STRING

### Example

This example prints "Hi" using the ASCII values of the characters "H" and "i".

```
print chr(72); chr(105)
```

## CLOSE Statement

Closes a file or files opened with the OPEN statement.

### Syntax

```
CLOSE [[#]filenumber[, [#]filenumber]...]
```

### Remarks

*filenumber* is the number of an open file. The file associated with the given file number will be closed.

CLOSE without any parameters will close all open files.

Although *QmodemPro for Windows* will automatically close files when the script terminates, you should always close any files that you open. There are only a limited number of file numbers available; if you run out of file numbers you will not be able to open any more files.

### See also

OPEN, RESET

### Example

This file opens a new file called "test.txt" for output, writes a line of test data to the file, then closes the file.

```
open "test.txt" for output as #1
print #1, "test data"
close #1
```

## CLS Statement

Clears the terminal screen and returns the cursor to the top left corner. This command also resets the current text color to the default text color.

### Syntax

```
CLS
```

### Remarks

This command is similar to the DOS CLS "clear screen" command.

### See also

COLOR

### Example

This example clears the screen and writes "Hello, world!" in the upper left hand corner.

```
cls
print "Hello, world!"
```

## COLOR Statement

Sets the current terminal color to the specified foreground and background.

### Syntax

```
COLOR foreground [, background]
```

### Remarks

Sets the current terminal color to the specified foreground and background values. If the background value is not specified, then it remains unchanged.

The color values used by this command are as follows:

| Value | Color | Value | Color |
|---|---|---|---|
| 0 | Black | 8 | Silver |
| 1 | Maroon | 9 | Red |
| 2 | Green | 10 | Lime |
| 3 | Olive | 11 | Yellow |
| 4 | Navy | 12 | Blue |
| 5 | Purple | 13 | Fuschia |
| 6 | Teal | 14 | Aqua |
| 7 | Gray | 15 | White |

### See also

CLS, SCREEN

## Example

This example sets the color to yellow on blue and prints a text string in that color.

```
color 11, 4
print "yellow text on blue background"
```

## CONFIGCAPTUREFILE Function

### Syntax

```
CONFIGCAPTUREFILE
```

### Remarks

This function returns the exact text contained in the Options/Files/File definitions dialog entry for the capture file name. This is usually the fully qualified path and filename of the capture file, but may return only the filename if no path was entered in the config dialog.

### See also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

### Example

```
UPLOAD (CONFIGCAPTUREFILE, Zmodem)
```

## CONFIGDOWNLOADPATH Function

### Syntax

```
CONFIGDOWNLOADPATH
```

### Remarks

This function returns the exact text contained in the Options/Files/Path definitions dialog entry for the download path. This is the fully qualified path for the download directory, without a trailing backslash.

### See also

CONFIGCAPTUREFILE, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

### Example

```
CHDIR CONFIGDOWNLOADPATH
```

## CONFIGLOGFILE Function

### Syntax

```
CONFIGLOGFILE
```

### Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the logfile name. This is usually the fully qualified path and filename of the log file, but may return only the filename if no path was entered in the config dialog.

### See also

CONFIGDOWNLOADPATH, CONFIGCAPTUREFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

### Example

```
UPLOAD (CONFIGLOGFILE, Zmodem)
```

## CONFIGSCRIPTPATH Function

### Syntax

```
CONFIGSCRIPTPATH
```

### Remarks

The function returns the exact text contained in the Options/Files/Path definitions dialog entry for the scripts path. This is the fully qualified path for the scripts directory, with no trailing backslash.

### See also

CONFIGDOWNLOADPATH, CONFIGCAPTUREFILE, CONFIGLOGFILE, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

### Examples

```
CHDIR CONFIGSCRIPTPATH
```

## CONFIGSCROLLBACKFILE Function

### Syntax

```
CONFIGSCROLLBACKFILE
```

### Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the scrollback filename. This is usually the fully qualified path and filename of the scrollback file, but may return only the filename if no path was entered in the config dialog.

### See also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGCAPTUREFILE, CONFIGTRAPFILE, CONFIGUPLOADPATH

### Example

```
UPLOAD (CONFIGSCROLLBACKFILE, Zmodem)
```

## CONFIGTRAPFILE Function

### Syntax

```
CONFIGTRAPFILE
```

### Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the trap filename. This is usually the fully qualified path and filename of the trap file, but may return only the filename if no path was entered in the config dialog.

### See also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGCAPTUREFILE, CONFIGUPLOADPATH

### Example

```
UPLOAD (CONFIGTRAPFILE, Zmodem)
```

## CONFIGUPLOADPATH Function

### Syntax

```
CONFIGUPLOADPATH
```

### Remarks

The function returns the exact text contained in the Options/Files/File definitions dialog entry for the upload file area path. This is the fully qualified path for the upload path, with no trailing backslash.

### See also

CONFIGDOWNLOADPATH, CONFIGLOGFILE, CONFIGSCRIPTPATH, CONFIGSCROLLBACKFILE, CONFIGTRAPFILE, CONFIGCAPTUREFILE

### Examples

```
CHDIR CONFIGUPLOADPATH
```

## CONST Statement

Used to assign symbolic names that will be used in place of actual values.

### Syntax

```
CONST name = expression [, name = expression ...]
```

### Remarks

*name* is the name of the new constant.

*expression* is the value to assign to the symbolic constant name.

The type of name is determined by the type of the expression.

Constants defined in subroutines or functions can only be used within the subroutine or function. Constants defined with a CONST statement in the main program body can be used throughout the program.

### Example

This example declares the constant "myname" and assigns a value to it.

```
const myname = "John Doe"
print "My name is "; myname
```

## COPYFILE Function

Copy a file to another directory or drive.

### Syntax

**CopyFile(***SourceFile, TargetFile***)**

### Remarks

The **CopyFile** function does just as the name describes, it copies a file from one location to another. It will issue no warning, prior to over-writing a file with an identical name, if one exists in the target directory. The function will return either **True** or **False** upon the success of the operation. Both *SourceFile* and *TargetFile* must include the file name as a path-only in *TargetName* will be rejected.

### See also

NAME

### Example

```
Rem Copy a file across drives

If CopyFile("C:WCLIST.OUT", "D:\BAK\DATA\WCLIST.BAK") Then
   Print "File copied"
Else
   Print "Copy failed"
   Beep
   WaitEnter
End If
```

## COS Function

Returns the cosine of an angle.

### Syntax

```
COS(angle)
```

### Remarks

*angle* is the measurement of an angle expressed in radians. You can convert radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

### See also

ATN, SIN, TAN

### Example

This example prints the cosine of 1 radian.

```
print cos(1)
```

## CSRLIN Function

Returns the current vertical coordinate position (row number) of the cursor.

### Syntax

```
CSRLIN
```

### Remarks

This value is usually an integer in the range 1 through 25, but may be larger depending on the number of lines set in the **Options/Emulations** dialog.

### See also

LOCATE, POS

### Example

This example clears the screen, then prints the cursor line twice. The first time it will be 1 since the cursor is on the top line of the screen, and the second time it will be 2 (the cursor moved down because of the first print statement).

```
cls
print csrlin
print csrlin
```

## CURDIR Function

Returns the current drive and directory.

### Syntax

```
CURDIR
```

### Remarks

The current directory of the current drive is returned without a trailing backslash (unless the current directory is the root directory).

### See also

CHDIR, CHDRIVE, CURDRIVE, MKDIR, RMDIR

### Example

This example prints the current directory for the current drive.

```
print "The current directory is "; curdir
```

## CURDRIVE Function

Returns the current drive letter.

### Syntax

```
CURDRIVE
```

### Remarks

The current drive letter is returned as an uppercase letter. This is the same drive that is returned by the CURDIR function.

### See also

CHDIR, CHDRIVE, CURDIR, MKDIR, RMDIR

### Example

This example prints the current drive letter.

```
print "The current drive is "; curdrive
```

## DATE Function

Returns the current date as a string.

### Syntax

```
DATE
```

### Remarks

The date is returned in the format specified in Windows Control Panel, International section, short date format.

### See also

TIME

### Example

This example prints today's date.

```
print "Today is "; date
```

## DECLARE Statement

Allows you to declare subroutines and functions before their actual definition. Also allows declaration of DLL subroutines and functions.

### Syntax

```
DECLARE [FUNCTION | SUB] name [LIB "libname" [ALIAS "aliasname"]]
[(argument list)] [AS returntype]
```

### Remarks

*name* is the name of the subroutine or function.

The argument list lists the parameters to the subroutine or function. This argument list must match the argument list declared in the actual definition of the subroutine or function.

The LIB and ALIAS clauses are used to declare a function that actually exists in another DLL. For more information on declaring and using DLL functions, see Using DLL Functions.

The final AS clause is used to declare the return type of a function.

Functions must be declared before they can be used. If the function definition appears later in the script source file from where you want to call the function, the DECLARE statement can be used to declare the function before the call.

### See also

CALL, FUNCTION, SUB

### Example

This example uses the DECLARE statement to declare a function so it can be used before its actual definition appears later in the file.

```
declare function timestwo(x as integer)
print timestwo(5)

function timestwo(x as integer)
  timestwo = x * 2
end function
```

## DEL Statement

Deletes a file from disk.

### Syntax

```
DEL filename
```

### Remarks

*filename* is the name of the file to delete. DOS wildcards * and ? are not supported.

This statement is identical to the script KILL command.

You cannot delete an open file, whether it is has been opened by your script, *QmodemPro for Windows*, or another application.

### See also

KILL, RMDIR

### Example

This example deletes the file "test.dat" from the current directory.

```
del "test.dat"
```

## DELAY Statement

Used to suspend script execution for a certain time interval.

### Syntax

```
DELAY time
```

### Remarks

*time* is the amount of time to suspend execution, expressed in seconds. If you want to delay for less than a second, use ordinary decimal notation.

This command is identical to the script PAUSE command.

### See also

PAUSE, WAITFOR, WHEN QUIET, WHEN TIME

### Example

This example uses the "atz" command to reset a Hayes-compatible modem, waits for half a second, then sends a command to dial a telephone number.

```
send "atz"
delay 0.5
send "atdt5551212"
```

## DEVICE Statement

Changes the currently active device.

### Syntax

```
DEVICE devicename
```

### Remarks

The device list is searched for *devicename*. If a matching device is found it is activated. If no matching device is found then the current device remains unchanged.

The special device name "-Default-" switches to the default device, and the device name "-Current-" switches to the current device (switching to the current device only serves to reinitialize the communications port).

### See also

DIAL, SETCOMM

### Examples

This example switches to the device "USR HST 14400" and sends a command to initialize the modem.

```
device "USR HST 14400"
send "atz"
```

## DIAL Statement

This statement is used to dial an entry or entries from the phonebook.

### Syntax

```
DIAL ENTRY number
or
DIAL GROUP groupname
or
DIAL SEARCH string
or
DIAL MANUAL number
```

### Remarks

There are four forms to the DIAL command:

#### DIAL ENTRY

This form allows you to dial a specific entry number in the phonebook.

#### DIAL GROUP

This form allows you to dial all the entries in a named group.

#### DIAL SEARCH

This form allows you to search for a string and dial all entries that contain that string.

#### DIAL MANUAL

This form allows you to dial a specific phone number from the script.

After dialing a number from a script in this way, the script file specified in the dialing directory (if any) is not executed. After connect the script proceeds from the next statement after the DIAL command.

## See also

ADDENTRY, DIALNEXT, HANGUP, LASTCONNECTPASSWORD, LASTCONNECTUSERID

## Examples

This example shows each of the three ways the DIAL command can be used.

```
dial entry 3
dial group "Morning mail"
dial search "Mustang"
dial manual "555-1212"
```

## DIALNEXT Function

This function is used to dial the next entry in a group after using the DIAL GROUP command.

### Syntax

```
DIALNEXT
```

### Remarks

When using the DIAL GROUP command, all the entries in the phonebook corresponding to the dialed group are marked for dial. After connecting to an entry, use the DIALNEXT function to continue dialing the remaining marked entries in the phonebook.

This function returns FALSE if there are no further marked entries in the phonebook, otherwise it returns TRUE.

### See also

ADDENTRY, DIAL, HANGUP

### Examples

This example shows how the DIALNEXT command is used in conjunction with DIAL GROUP.

```
dial group "Morning mail"
do
  ... do whatever needs to be done online ...
loop while dialnext
```

## DIALOG Statement

This statement is used to declare a dialog box type.

### Syntax

```
DIALOG dialogtype x, y, w, h
  [CAPTION caption]
  [FONT size, fontname]
  [integer-field AS CHECKBOX title, id, x, y, w, h]
  [integer-field AS COMBOBOX id, x, y, w, h]
  [CTEXT title, id, x, y, w, h]
  [DEFPUSHBUTTON title, id, x, y, w, h]
  [string-field AS EDITTEXT id, x, y, w, h]
  [GROUPBOX title, id, x, y, w, h]
  [integer-field as LISTBOX id, x, y, w, h]
  [LTEXT title, id, x, y, w, h]
  [PUSHBUTTON title, id, x, y, w, h]
  [integer-field AS RADIOBUTTON title, id, x, y, w, h]
  [RTEXT title, id, x, y, w, h]
  ...
END DIALOG
```

### Remarks

The DIALOG statement declares a *dialog template*. A dialog template is much like a user defined type in that it contains named fields in which you can place information. Dialog templates can also have a number of unnamed fields which serve to place extra information in the Windows dialog that is created based on this template.

Dialog boxes are discussed in depth in the section titled Using Dialog Boxes.

## See also

DIALOGBOX

## DIALOGBOX Function

Displays and executes a dialog box based on a dialog box template.

### Syntax

```
DIALOGBOX(dialogvar)
```

### Remarks

This function creates and executes a dialog box based on the dialog box variable. Its return value depends on the event that caused the dialog box to close. In most cases this will be either IDOK or IDCANCEL depending on whether the user pressed the OK button or the Cancel button to close the dialog box. In more advanced cases the meaning of the return value will be user defined.

The dialog box variable used in this function should not be already active.

Dialog boxes are discussed in depth at the end of this chapter, in the section titled Using Dialog Boxes.

### See also

DIALOG

## DIM Statement

The DIM statement is used to declare variables of any type including array types. An array is a variable containing a series of values that are all of the same type.

### Syntax

```
DIM var[([lowerbound TO] upperbound)] AS type[, var[([lowerbound TO]
upperbound)] AS type...]
```

### Remarks

var is the name of the array or variable being declared.

lowerbound and upperbound declare the lowest and highest subscript values that are allowed if an array is being declared. If lowerbound is omitted, it defaults to zero.

Each variable must have an associated type declaration with the AS clause.

Variables declared within a subroutine or function are only available from within that subroutine or function. Variables declared in the main program body are available throughout the entire script file.

An array consists of a number individual variables, called "elements", which are referred to by numbers, called "subscripts" indicating the position of each element in the array.

### See also

STATIC

## Example

This example declares a simple integer "i" and an array "a" which refers to a sequence of six integers, numbered a(0) through a(5).

```
dim i as integer, a(5) as integer
for i = 1 to 5
  a(i) = i*2
next
for i = 1 to 5
  print a(i)
next
```

## DO ... LOOP Statement

Repeatedly executes a block of statements while (or until) a specified condition is met.

### Syntax

```
DO [{WHILE | UNTIL} expression]
  [statements]
LOOP
```

or

```
DO
  [statements]
LOOP [{WHILE | UNTIL} expression]
```

### Remarks

The first example above tests for the specified condition at the beginning of the loop, and exits the loop when the condition is met.

The second example tests for the specified condition at the end of the loop, and continues until the condition is met.

*expression* is any logical expression that evaluates to either true (nonzero) or false (zero). The keyword WHILE repeats the loop while the expression remains true. The UNTIL keyword repeats the loop until the expression becomes true.

*statements* are program statements that are repeated. Note that statements are optional; a loop can be empty, waiting for an external event such as a keypress or an incoming character.

Every DO statement in a program must have a corresponding LOOP, and each LOOP must have a preceding DO.

### See also

EXIT DO, FOR ... NEXT, WHILE ... WEND

## Example

This example prints out the integers from 1 through 5 using a DO WHILE ... LOOP statement.

```
dim i as integer
i = 0
do while i < 5
  i = i + 1
  print i
loop
```

## DOORWAY Function

Used to get or set the current state of the Doorway toggle.

### Syntax

```
DOORWAY
or
DOORWAY(onoff)
```

### Remarks

There are two forms to the DOORWAY function. The first form takes no arguments and simply returns the current setting. The second form sets the state of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means Doorway mode is turned on. A zero value means Doorway mode is turned off.

When turned on, Doorway mode causes all keystrokes that can be typed on the keyboard to be sent directly to the remote. This will work only with a remote host that understands the Doorway keystrokes.

### Example

This example shows how to save the doorway setting, change it, and then restore it..

```
dim olddoorway as integer
olddoorway = doorway(on)
  ...
doorway olddoorway
```

## DOSVERSION Function

Returns the current DOS version as a string.

### Syntax

```
DOSVERSION
```

### Remarks

This function returns the DOS version as a string in the format X.Y. X is the major version number and Y is the minor revision number.

For example, DOS version 6.0 would be reported as "6.0".

### See also

VERSION, WINVERSION

### Example

```
print "Currently using DOS "; dosversion
```

## DOWNLOAD Function

Used to receive files from a remote computer.

### Syntax

```
DOWNLOAD(filename, protocol)
```

### Remarks

This function initiates a file transfer to receive files from a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

The protocol is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT, BPLUS

For the first five protocols, the filename parameter must specify an actual file name in which to place the received file. The ASCII and Xmodem variant protocols do not supply a filename, so one must be supplied in the DOWNLOAD function.

For the last five protocols, the filename parameter should be the name of a directory in which to place the received files. If this parameter is an empty string (" ") then the download directory specified in **Options/Files/Path Definitions** will be used.

The DOWNLOAD function returns zero if the file transfer was successful. If the transfer was unsuccessful, DOWNLOAD returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the RECEIVEFILE function.

### See also

RECEIVEFILE, SENDFILE, UPLOAD

## Example

This example receives a file from a remote computer, assuming that the remote computer has already started to send the file.

```
if download("", Zmodem) = 0 then
  print "file transfer ok!"
end if
```

## DUPLEX Function

Used to get or set the current duplex (local echo) setting.

### Syntax

```
DUPLEX
or
DUPLEX(onoff)
```

### Remarks

There are two forms to the DUPLEX function. The first form takes no arguments and simply returns the current duplex setting. The second form sets the duplex and returns the previous state of the duplex setting.

In both the parameter and the return value, a nonzero value means full duplex or no local echo. A zero value means half duplex or local echo.

### See also

HOSTECHO

### Example

This example shows how the DUPLEX function can be used to control the local echo of characters sent to the communications port.

```
dim oldduplex as integer
oldduplex = duplex(0)
send "this will be echoed locally"
duplex oldduplex
send "this will not be echoed locally"
```

## EDITFILE Statement

Used to invoke the internal editor with a specified file.

### Syntax

```
EDITFILE filename
```

### Remarks

This function brings up the internal editor with the named file loaded ready for editing. Note that that script continues to run after the editor is started.

### See also

VIEWFILE

### Example

This example shows how the EDITFILE command might be used to edit the host mode user file.

```
editfile "host.usr"
```

## ELSE Statement

The ELSE statement is used to introduce the alternative portion of an IF statement. See the discussion of the IF statement for more information.

## ELSEIF Statement

The ELSEIF statement is used to introduce an optional alternative portion of an IF statement. See the discussion of the IF statement for more information.

## EMULATION Statement

This statement is used to change the current terminal emulation.

### Syntax

```
EMULATION emulation
```

### Remarks

The emulation argument must be one of the following predefined constants:

ADDSVP60, ADM3A, ANSI, AVATAR, DEBUGASCII, DEBUGHEX, DG100, DG200, DG210, HAZELTINE1500, HEATH19, IBM3101, IBM3270, RIPSCRIP, TTY, TVI910, TVI912, TVI920, TVI922, TVI925, TVI950, TVI955, VIDTEX, VT100, VT102, VT220, VT320, VT52, WYSE30, WYSE50, WYSE60, WYSE75, WYSE85, WYSE100, WYSE185

### See also

DEVICE, DIAL

### Example

This example changes the current terminal emulation to RIPscrip.

```
emulation ripscrip
```

## END Statement

Ends a script and returns to normal terminal operation.

### Syntax

```
END
```

### Remarks

The END statement immediately terminates the currently executing script and returns to normal terminal operation. It may be executed at any time. Any currently open files are closed before the script ends.

### See also

STOP

### Example

This example print a string to the terminal, then ends the script. The second print statement is never executed.

```
print "hello world"
end
print "this is never executed"
```

## ENVIRON Function

Returns the value of the specified DOS environment variable.

### Syntax

```
ENVIRON(envname)
```

### Remarks

*envname* is the name of the environment variable to retrieve. The *envname* parameter should be in upper case. If the specified environment variable is not found, an empty string will be returned.

### Example

This example prints out the current DOS PATH setting.

```
print environ("PATH")
```

## EOF Function

Used to determine whether the end of a file has been reached.

### Syntax

```
EOF(filenum)
```

### Remarks

This function returns TRUE if the end of the file specified by the *filenum* parameter has been reached.

### See also

LOC, LOF, SEEK

### Example

This example prints the contents of a text file line by line until the end of the file is encountered.

```
dim a as string
open "test.dat" for input as #1
do while not eof(1)
  input #1, a
  print a
loop
close #1
```

## EQV Operator

EQV performs a bitwise equivalence operation between its operands.

### Syntax

```
op1 EQV op2
```

### Remarks

op1 and op2 are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the EQV operator:

| a | b | a EQV b |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### See also

AND, IMP, NOT, OR, XOR

### Example

This example shows how the EQV operator can be used in an IF statement.

```
dim i as integer, j as integer, k as integer
i = 10
j = 8
k = 6
if i > j eqv j > k then
  print "both expressions are true or both are false"
end if
```

## ERROR Statement

Causes a predefined or user defined runtime error to occur.

### Syntax

```
ERROR errvalue
```

### Remarks

If there is a CATCH handler for the error value, then it will be executed. Otherwise, the script will be terminated automatically. See the CATCH statement for a list of predefined runtime error values.

### See also

CATCH

### Example

This example forces the ERR_FILEOPEN error to occur, which will cause the script to jump immediately to the handler for ERR_FILEOPEN. As a result, the first print statement won't be executed.

```
error err_fileopen
print "this won't be printed"

catch err_fileopen
print "caught the error"
```

## EXISTS Function

Determines whether a specific file exists on disk.

### Syntax

```
EXISTS(filename)
```

### Remarks

This function returns TRUE if the specified file exists on disk. DOS wildcards are not accepted.

### See also

FINDFIRST, FINDNEXT

### Example

This example determines whether the "c:\autoexec.bat" file still exists.

```
if exists("c:\autoexec.bat") then
  print "autoexec.bat is still there"
end if
```

## EXIT Statement

Aborts a loop, subroutine, or procedure without waiting for normal termination or return.

### Syntax

```
EXIT [DO | FOR | FUNCTION | SUB]
```

### Remarks

One of the four keywords above must be included in the EXIT statement.

Use the EXIT DO and EXIT FOR statements to exit a loop defined by a DO ... LOOP or a FOR ... NEXT statement.

If a loop is nested, the EXIT statement will stop only the current (innermost) loop.

Use the EXIT FUNCTION and EXIT SUB statements to abort procedures defined by a FUNCTION or SUB statement.

If you are exiting a function with EXIT FUNCTION and have not already assigned a value to the function result, the function will return zero or an empty string, depending on the type of the function.

### See also

DO ... LOOP, FOR ... NEXT, FUNCTION, SUB

### Example

This example uses the EXIT DO statement to break out of a loop when a certain condition is satisfied.

```
dim i as integer
i = 0
do
  i = i + 1
  if i > 10 then exit do
  print i
loop
```

## EXP Function

Returns the natural antilogarithm of a value.

### Syntax

```
EXP(number)
```

### Remarks

This function returns e (the natural logarithm base) raised to the power of *number*.

### See also

LOG

### Example

This example raises e to the power of 2 (in other words, it computes the square of e) and prints the result.

```
print exp(2)
```

## FINDFIRST Function

Used to find the first of a set of files matching a wildcard filename specification.

### Syntax

```
FINDFIRST(name, attr, sr)
```

### Remarks

*name* is the filename specification to search for, and can include DOS wildcard characters * and ?.

*attr* is the file attribute to look for. To search for normal files use 0 for this parameter.

*sr* is a variable of type SEARCHREC that will hold the file information for the file found.

This function is normally used in conjunction with the FINDNEXT function to gather a list of files.

The declaration for the SEARCHREC type is as follows:

```
type searchrec
  attribute as byte
  time as integer
  date as integer
  size as long
  name as string*13
end type
```

The date and time are encoded using the following formula:

```
time = (hours * 2048) + (minutes * 32) + (seconds / 2)
date = ((year - 1980) * 512) + (month * 32) + day
```

This function returns zero if a file matching the specification was found. It returns a nonzero value if no files were found.

## See also

EXISTS, FINDNEXT

## Example

This example lists all the files in the "c:\qmwin" directory.

```
dim sr as searchrec, r as integer
r = findfirst("c:\qmwin\*.*", 0, sr)
do while r = 0
  print sr.name
  r = findnext(sr)
loop
```

## FINDNEXT Function

Finds the next of a set of files that was started using the FINDFIRST function.

### Syntax

```
FINDNEXT(sr)
```

### Remarks

*sr* is a variable of type SEARCHREC.

This function is always used in conjunction with the FINDFIRST function to search for a list of files.

This function returns zero if another file matching the specification was found. It returns a nonzero value if no more files were found.

### See also

EXISTS, FINDFIRST

### Example

See the FINDFIRST function for an example of how to use this function.

## FIX Function

Returns the integer or whole number portion for a numeric expression.

### Syntax

```
FIX(number)
```

### Remarks

FIX and INT operate the same for positive values, however with negative values their operation is slightly different. FIX returns the next-higher integer with negative values, while INT returns the next lower integer with negative values.

### See also

INT

### Example

This example shows how the FIX function rounds negative numbers up to the next higher integer. In this example the output will be 3 and -4.

```
print fix(3.7), fix(-4.9)
```

## FLUSH Statement

Flushes either or both of the input and output communications buffers.

### Syntax

```
FLUSH [INPUT] [OUTPUT]
```

### Remarks

The INPUT keyword indicates that the communications input buffer should be flushed.

The OUTPUT keyword indicates that the communications output buffer should be flushed.

### See also

BREAK, HANGUP

### Example

This example shows how the FLUSH INPUT statement might be used to get rid of stray input data caused by hanging up the modem.

```
hangup
delay 2
flush input
```

## FOR ... NEXT Statement

Executes a sequence of program statements a specified number of times.

### Syntax

```
FOR index = initial TO final [STEP step]
  [statements]
NEXT [index]
```

### Remarks

*index* is a numeric variable of type byte, integer, long, or real, and counts the number of times the loop executes.

*initial* is the initial value assigned to the counter at the beginning of the loop.

*final* is the ending value against which the loop tests the variable index for each pass.

*step* is the amount by which the counter is incremented after each pass in a FOR ... NEXT loop. The default value is 1.

When the value of index is higher than the value of final, the loop finishes, and returns control to the statement following the keyword NEXT. If the step value is less than zero, then the index is compared against the final value and the loop is stopped if *index* is less than *final*.

### See also

DO ... LOOP, EXIT, WHILE ... WEND

### Example

This example prints "hello" ten times.

```
dim i as integer
for i = 1 to 10
  print "hello "; i
next
```

## FREEFILE Function

Returns the next available file number.

### Syntax

```
FREEFILE
```

### Remarks

You can assign file numbers on the fly by storing the FREEFILE result in a variable and passing the variable to OPEN and CLOSE statements.

### Return value

This function returns the first available file number (one that does not refer to a currently open file). If no file number is available, then the return value is −1.

### See also

OPEN

### Example

This example opens a file and outputs some test data to it. It doesn't matter how many files are currently open because it uses the FREEFILE function to find a free file number.

```
dim f as integer
f = freefile
open "test.dat" for output as #f
print #f, "this is a test"
close #f
```

## FUNCTION Statement

Allows you to define your own subprograms that return a value to the caller.

### Syntax

```
FUNCTION name[(arg AS type[, arg AS type]...)] AS type
  ...
  name = expr
  ...
END FUNCTION
```

### Remarks

*name* is the name you assign to the function.

*arg* is the name of a formal argument to the function. Each argument must have an associated type declaration using the AS keyword.

You may define local variables within your user-defined function. You can use STATIC declarations to preserve the value of individual variables across function calls.

To return a value from the function, use an assignment to the name of the function. The value you assign to the function is remembered until the function ends, at which point it is retrieved and returned to the caller of the function.

Functions cannot be defined within another function definition.

### See also

DECLARE, EXIT, STATIC, SUB

## Example

This example declares a function that returns its argument multiplied by two and incremented by one.

```
function f(x as integer) as integer
  f = x*2 + 1
end function

print f(2)
print f(f(7))
```

## GET Statement

Reads information from a random-access or binary file into a record variable.

### Syntax

```
GET [#]filenumber, [position], variable
```

### Remarks

*filenumber* is the number assigned to an open file.

*position* is the number of the record in a random access file, or the number of the byte in a binary file. Note that unlike some other languages, the first record or byte in the file is number 1, not number 0. If the position is not specified, then the record is read from the current file position.

*variable* is the name of the variable that receives the returned data. WARNING: If you are reading a record from a random access file, you must ensure that the variable is of the correct type, otherwise unpredictable results may occur.

### See also

INPUT, OPEN, PUT, TYPE ... END TYPE

### Example

This example creates a random access file, writes a record to it from the variable d, and reads it back into the variable z. It then prints out the contents of z to make sure that the operation succeeded.

```
type daterec
  day as integer
  month as integer
  year as integer
end type
dim d as daterec, z as daterec
d.day = 11
d.month = 10
d.year = 1993
open "test.dat" for random as #1 len = len(daterec)
put #1, 1, d
get #1, 1, z
close #1
print z.day, z.month, z.year
```

## GOSUB ... RETURN Statement

Branches unconditionally to the specified line number or label to execute a subroutine.

### Syntax

```
GOSUB {line|label}
    .
label:
    .
RETURN
```

### Remarks

*line* and *label* indicate the line number or label at which the subroutine should begin executing.

The RETURN statement resumes execution at the statement following the original GOSUB statement.

You cannot use GOSUB to branch into or out of a sub-program or user-defined function.

The SUB and FUNCTION constructs provide a much better method of supporting subprograms than GOSUB and RETURN.

### See also

FUNCTION, GOTO, SUB

### Example

This example has a very simple subroutine called "printit" that prints the value of i and returns.

```
dim i as integer
i = 5
gosub printit
i = 9
gosub printit
end
printit:
print i
return
```

## GOTO Statement

Branches unconditionally to a specified line or label.

### Syntax

```
GOTO {line|label}
```

### Remarks

Program execution continues at the referenced line or label, or the first executable statement immediately following the line or label.

You cannot use GOTO to branch into or out of a sub-program or user-defined function.

Proponents of structured programming recommend against using GOTO statements wherever IF statements and DO loops can accomplish the same task, on the grounds that structured code is more efficient and easier to maintain.

### See also

FUNCTION, GOSUB, SUB

### Example

This example demonstrates a simple loop structure using a GOTO statement.

```
dim i as integer
i = 0
again:
i = i + 1
if i < 10 then goto again
```

## HANGUP Statement

Hangs up the modem by sending the modem hangup string, without asking for confirmation.

### Syntax

```
HANGUP
```

### Remarks

This statement sends the modem hangup string as defined in **Options/Device/Modem**. If the hangup string indicates that DTR be dropped or a BREAK sent, this command will do so.

### See also

BREAK, CARRIER

### Example

This example sends a command to log off an on-line service, waits five seconds, then hangs up the modem.

```
send "bye"
delay 5
hangup
```

## HEX Function

Converts an integer or long integer expression to a hexadecimal (base 16) value.

### Syntax

```
HEX(number)
```

### Remarks

*number* is the numeric variable or expression that HEX converts. The value may be any numeric type — it will be converted to an integer or a long integer. Fractional values are ignored.

### See also

OCT

### Example

This example prints the hexadecimal representation of the number 42.

```
print hex(42)
```

## HOSTECHO Function

Used to turn on and off character echo conventions that are used with the host mode.

### Syntax

```
HOSTECHO [ON | OFF]
```

### Remarks

This command modifies the operation of *QmodemPro* in the following ways:

Incoming data is not displayed on the screen, written to the capture file, or sent to the printer. It is only sent to the script program for processing.

Data that is output to the communications port is automatically echoed to the terminal screen.

When using the SEND command to output data to the communications port, a carriage return/line feed pair is sent at the end of the line if no semicolon is present (this is in contrast to the standard behavior of sending just a carriage return).

### See also

DUPLEX

### Example

This example shows what will happen when HOSTECHO is turned on. For the best example of how this command works, see the host mode script HOST.SCR.

```
dim name as string
hostecho on
send "What is your first name? ";
```

## IF Statement

Makes a decision regarding program flow, based on the result returned by an expression.

### Syntax

```
IF expression THEN then-part ELSE else-part
or
IF expression1 THEN
   statements-1
[ELSEIF expression2 THEN
   statements-2]
[ELSE
   statements-n]
END IF
```

### Remarks

*expression* is an expression that yields a result of true (nonzero) or false (zero).

*then-part* is a set of statements to be executed if *expression* is true.

*else-part* is a set of statements that will be executed if *expression* is false.

ELSE and ELSEIF are optional. ELSE allows you to execute a set of statements if *expression* is false.

ELSEIF allows you to test for several different conditions within a single IF statement. In the construct above, *expression2* would only be tested if *expression1* was false.

### See also

DO ... LOOP, SELECT CASE

## Example

This example demonstrates both types of IF statement syntax.

```
dim i as integer
i = 5
if i = 7 then print "is seven" else print "not seven"
if i = 5 then
  print "i is 5"
elseif i = 9 then
  print "i is 9"
else
  print "i is neither 7 nor 9"
end if
```

## IMP Operator

IMP performs a bitwise implication operation between its operands.

### Syntax

```
op1 IMP op2
```

### Remarks

*op1* and *op2* are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the IMP operator:

| a | b | a IMP b |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### See also

AND, EQV, NOT, OR, XOR

### Example

This example prints out the truth table above.

```
dim a as integer, b as integer
print "a", "b", "a IMP b"
for a = 0 to 1
  for b = 0 to 1
    print a, b, a IMP b
  next
next
```

## $INCLUDE Directive

Includes another source file into the current source file.

### Syntax

```
'$INCLUDE 'filename'
```

### Remarks

The $INCLUDE directive is an instruction to the script compiler to include the named script file into the current file. This is useful if you have a standard set of subroutines and functions that you want to share between multiple scripts.

### Example

```
[in file a.scr]
'$include 'b.scr'
call test

[in file b.scr]
sub test
  print "Hello, world!"
end sub
```

## INKEY Function

Reads a character from the keyboard, communications port, or a file.

### Syntax

```
INKEY(filenumber)
```

### Remarks

If there is no character ready when reading from the keyboard or communications port, this function returns an empty string.

When reading from a file or communications port, or if there is an ordinary character waiting from the keyboard, this function returns a single-character string containing the character.

When reading from the keyboard, the following strings are returned if the corresponding special key is pressed:

```
"Tab", "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10",
"F11", "F12", "Backspace", "Enter", "Scroll Lock", "Pause", "Gray
Insert", "Gray Delete", "Gray Home", "Gray End", "Gray Pageup", "Gray
Pagedown", "Gray Up", "Gray Down", "Gray Left", "Gray Right",
"Numlock", "Pad /", "Pad *", "Pad -", "Pad +", "Pad Enter", "Pad .",
"Pad 0", "Pad 1", "Pad 2", "Pad 3", "Pad 4", "Pad 5", "Pad 6", "Pad
7", "Pad 8", "Pad 9", "Pad Delete", "Pad Insert", "Pad End", "Pad
Down", "Pad Pagedown", "Pad Left", "Pad Clear", "Pad Right", "Pad
Home", "Pad Up", "Pad Pageup"
```

### See also

INPUT

### Example

This example reads a single keypress from the keyboard and prints the resulting string.

```
dim c as string
do
  c = inkey
loop until c <> ""
print "key pressed: "; c
```

## INPUT Statement

Captures a line of data from the keyboard, communications port, or a file, and places the data into a string variable.

### Syntax

```
INPUT [#filenum,] variable
```

### Remarks

*filenum* is the file number to use.

*variable* is the variable name into which you want to place the captured data.

### See also

GET, INKEY

### Example

This example simply reads a line of input from the user and prints it back out again.

```
dim a as string
input a
print a
```

## INSTR Function

Searches for the first occurrence of a string of characters within a specified string.

### Syntax

```
INSTR([start,] string1, string2)
```

### Remarks

*start* is the position at which to begin the search. This parameter is optional — the default is to start searching at position 1.

*string1* is the string to search.

*string2* is the text to search for within the string.

INSTR returns the position of the first occurrence of the specified text. If INSTR does not find the specified string, it returns a value of 0.

### See also

LEFT, MID, RIGHT

### Example

This example looks for the string "Windows" within the string *"QmodemPro for Windows"*.

```
dim a as string
a = "QmodemPro for Windows"
print instr(a, "Windows")
```

## INT Function

Returns the integer or whole number portion of a numeric expression.

### Syntax

```
INT(expr)
```

### Remarks

FIX and INT operate the same for positive values, however with negative values their operation is slightly different. FIX returns the next-higher integer with negative values, while INT returns the next lower integer with negative values.

### See also

FIX

### Example

This example shows how the INT function rounds negative numbers down to the next lower integer. In this example the output will be 3 and -5.

```
print int(3.7), int(-4.9)
```

## INTEGER Type

Used to declare a variable that can handle integer numbers.

### Remarks

Variables of integer type can hold values that range from -32768 to 32767.

### See also

BYTE, DIM, LONG, REAL, TYPE

### Example

This example declares a variable of type INTEGER and assigns a value to it.

```
dim i as integer
i = 5000
print i
```

## KILL Statement

Deletes a file from disk.

### Syntax

```
KILL filename
```

### Remarks

*filename* is the name of the file to delete. DOS wildcards * and ? are not supported.

You cannot delete an open file.

This statement is identical to the script DEL command.

### See also

DEL, RMDIR

### Example

This example removes the file "test.dat" from the current directory.

```
kill "test.dat"
```

## LASTCONNECTPASSWORD Function

. This function returns the current password.

### Syntax

```
LASTCONNECTPASSWORD
```

### Remarks

The password returned corresponds to the dialing directory entry that was used for the most recent connection.

### See also

DIAL, LASTCONNECTUSERID

### Example

```
waitfor "password?"
send lastconnectpassword
```

## LASTCONNECTUSERID Function

This function returns the current user id.

### Syntax

```
LASTCONNECTUSERID
```

### Remarks

The user id returned corresponds to the dialing directory entry that was used for the most recent connection.

### See also

DIAL, LASTCONNECTPASSWORD

### Example

```
waitfor "name:"
send lastconnectuserid
```

## LCASE Function

Returns a copy of a string with all upper case characters converted to lower case.

### Syntax

```
LCASE(string)
```

### Remarks

*string* can be any string expression.

This operation is useful for making case insensitive comparisons of text. The UCASE function operates similarly, but converts the specified text to upper case.

### See also

UCASE

### Example

This example demonstrates using the LCASE function to print a lower case version of a string.

```
dim a as string
a = "QmodemPro for Windows"
print lcase(a)
```

## LEFT Function

Returns a string consisting of a specified number of characters starting at the left (beginning) of a string.

### Syntax

```
LEFT(string, num)
```

### Remarks

*string* is any string expression.

*num* is any number in the range 0 through 32767.

If there are fewer than *num* characters in *string*, the entire string will be returned without any padding.

### See also

INSTR, MID, RIGHT

### Example

This example uses the LEFT function to print the first word in the string *"QmodemPro for Windows"*.

```
dim a as string
a = "QmodemPro for Windows"
print left(a, 9)
```

## LEN Function

Returns the number of characters in a string, or returns the size in bytes of a user defined type.

### Syntax

```
LEN(string)
or
LEN(typename)
```

### Remarks

*string* is any string expression. Used in this way the function will return the number of characters in the string.

*typename* is the name of any user defined type. Used in this way the function will return the number of bytes taken up by a variable of the indicated type. This is useful when used with the LEN = clause in the OPEN statement.

### See also

INSTR, LEFT, MID, OPEN, RIGHT

### Example

This example prints the length of the string *"QmodemPro for Windows"*. See the OPEN script command for an example of how to use LEN with OPEN.

```
dim a as string
a = "QmodemPro for Windows"
print len(a)
```

## LET Statement

Assigns the value of an expression to a variable.

### Syntax

```
[LET] var = expr
```

### Remarks

The LET keyword is an assignment statement — it allows you to assign values to variables.

You must declare a variable using DIM before you can assign a value to it.

The keyword itself is optional, so statements such as

```
LET tries = 1
```

is functionally identical to

```
tries = 1
```

The LET keyword is usually omitted in all cases.

### See also

DIM, TYPE

### Example

This example shows how to declare a variable, assign a value to it, and print its value on the screen.

```
dim i as integer
i = 5
print i
```

## LOADPHN Statement

Loads a new phonebook.

### Syntax

```
LOADPHN filename
```

### Remarks

This command loads a new phone book just like the phonebook's **File/Open** command would.

### See also

DIAL

### Example

This example shows how to load a new phone book and dial its first entry.

```
loadphn "test.phn"
dial entry 1
```

## LOC Function

Returns the current position of a pointer in an open file, indicating the point where the next read or write operation will take place.

### Syntax

```
LOC(filenumber)
```

### Remarks

*filenumber* is the number assigned to the open file.

If the file is opened for sequential or binary access, LOC returns the current file position in bytes.

If the file is opened for random access, LOC returns the current record number based on the record size specified in the LEN = clause in the OPEN statement.

### See also

GET, OPEN, PUT, SEEK

### Example

This example writes an integer to a file and reports the new file position. In this case the file position after writing will be 2.

```
dim i as integer
i = 5
open "test.dat" for random as #1 len = len(integer)
put #1, 1, i
print loc(1)
close #1
```

## LOCATE Statement

Positions the cursor on the screen.

### Syntax

```
LOCATE row, column
```

### Remarks

*row* is the number of the row at which the cursor is positioned. Rows are numbered 1 through the maximum number of rows on the screen.

*column* is the number of the column at which the cursor is positioned. Columns are numbered 1 through the number of columns on the screen.

### See also

CSRLIN, POS

### Example

This example places the cursor on row 5, column 5, and writes a string indicating that fact.

```
locate 5, 5
print "this is at position 5, 5"
```

## LOF Function

Returns the number of records in an open file.

### Syntax

```
LOF(filenumber)
```

### Remarks

*filenumber* is the number under which the file was opened.

If the file is opened for sequential or binary access, LOF returns the size of the file in bytes.

If the file is opened for random access, LOF returns the size of the file based on the record size specified in the LEN = clause in the OPEN statement.

### See also

LOC, OPEN

### Example

This example opens a file for sequential access and outputs the length of the file.

```
open "test.dat" for input as #1
print lof(1)
close #1
```

## LOG Function

Returns the natural logarithm of a number.

### Syntax

```
LOG(expression)
```

### Remarks

*expression* is any numeric expression.

A natural logarithm is the power to which the constant e (about 2.718282) must be raised to obtain a given number. This should not be confused with common logarithms, which are based on 10 rather than e.

If you try to take the logarithm of zero or a negative number, the ERR_MATH error will be generated. You can catch this error with the CATCH statement.

### See also

CATCH, EXP

### Example

This example prints the logarithm of the number 6.

```
print log(6)
```

## LOGFILE Statement

Turns on or off the log file.

### Syntax

```
LOGFILE {ON | OFF}
```

### Remarks

This command turns on or off the log file just like the **File/Log Toggle** command from the main terminal menu. The log file name is defined in **Options/Files/File Definitions**.

### See also

CAPTURE

### Example

This example turns on the log file, uploads a file, then turns it off again.

```
logfile on
call upload "c:\qmwin\test.dat", Zmodem
logfile off
```

## LONG Type

Used to declare a variable that can handle integer numbers larger than those handled by the INTEGER type.

### Remarks

Variables of the long type can hold values that range from -2147483648 to 2147483647.

### See also

BYTE, DIM, INTEGER, REAL, TYPE

### Example

This example declares a variable of type long, assigns a number to it, and prints out the number.

```
dim i as long
i = 500000
print i
```

## LOOP Statement

Used to mark the end of a DO ... LOOP statement. See the DO statement for more information.

## LTRIM Function

Trims leading spaces (spaces on the left) from a string expression.

### Syntax

```
LTRIM(string)
```

### Remarks

*string* is any string expression.

### See also

RTRIM

### Example

This example trims the leading spaces off the given string and prints the results within angle brackets.

```
dim a as string
a = "  QmodemPro for Windows  "
print ">"; ltrim(a); "<"
```

## MAXIMIZE Statement

Causes *QmodemPro for Windows* to maximize its application window on the Windows desktop.

### Syntax

```
MAXIMIZE
```

### Remarks

The *QmodemPro for Windows* application window is maximized and brought to the front of the Windows desktop.

### See also

ACTIVATE, MINIMIZE, MOVE, SIZE

### Example

This example causes the *QmodemPro for Windows* application to maximize its window.

```
MAXIMIZE
```

## MID Function

Retrieves a substring from an arbitrary position in another string.

### Syntax

```
MID(string, start, num)
```

### Remarks

*string* is any string expression.

*start* is the position of the first character to copy.

*num* is the number of characters to return from within the string expression.

If there are not enough characters in the string to supply enough characters in the result, a shorter string than that asked for will be returned.

### See also

LEFT, RIGHT

### Example

This example prints some characters from the middle of a given string.

```
dim a as string
a = "QmodemPro for Windows"
print mid(a, 7, 11)
```

## MINIMIZE Statement

Causes *QmodemPro for Windows* to minimize its application window on the Windows desktop. Minimizing an application makes it into an icon.

### Syntax

```
MINIMIZE
```

### Remarks

The *QmodemPro for Windows* application window is minimized and its icon is placed near the bottom of the desktop. Any other windows opened by *QmodemPro for Windows* (phonebook, GIF viewer, editor, and so on) will disappear until the application is restored.

### See also

ACTIVATE, MAXIMIZE, MOVE, SIZE

### Example

This example causes the *QmodemPro for Windows* application to minimize itself and place its icon at the bottom of the screen.

```
MINIMIZE
```

## MKDIR Statement

Creates a directory on the disk.

### Syntax

```
MKDIR directory
```

### Remarks

*directory* is the name of the directory to create.

When creating a new directory, each of the directories leading up to the last directory name must already exist. If you try to create the directory "c:\abc\def" and the directory "c:\abc" does not already exist, you will get an ERR_PATH error.

### See also

CHDIR, CURDIR, RMDIR

### Example

This example makes a new directory called "test" under the "c:\qmwin" directory.

```
mkdir "c:\qmwin\test"
```

## MOD Operator

Returns the remainder of a division between two integers.

### Syntax

```
op1 MOD op2
```

### Remarks

*op1* and *op2* can be of any integer numeric type (byte, integer, long).

If *op2* is zero the ERR_MATH error will be generated.

### Example

This example prints 144, which is the remainder after 400 is divided by 256.

```
print 400 mod 256
```

## MOUSECLICK Statement

Used to simulate a mouse click for the RIPscrip emulation.

### Syntax

```
MOUSECLICK x, y
```

### Remarks

This statement simulates a mouse click on an area of the screen when using the RIPscrip emulation. The x and y parameters are the horizontal and vertical coordinates of the mouse click. This command is generated by Quicklearn when using the RIPscrip emulation.

If the RIPscrip emulation is not active, this command has no effect.

### See also

EMULATION

### Example

This example simulates a mouse click at position 10, 10 on the screen.

```
mouseclick 10, 10
```

## MOVE Statement

Used to move the *QmodemPro for Windows* application window on the Windows desktop.

### Syntax

```
MOVE x, y
```

### Remarks

x is the horizontal coordinate of the new position of the window. X coordinate values range from 0 to the width of your screen minus one.

y is the vertical coordinate of the new position of the window. Y coordinate values range from 0 to the height of your screen minus one.

If you are using standard VGA with a 640x480 screen then the X values can range from 0 through 639 and Y values can range from 0 through 479.

### See also

ACTIVATE, MAXIMIZE, MINIMIZE, SIZE

### Example

This example moves the *QmodemPro for Windows* application window so its upper left hand corner is at position (300, 200).

```
MOVE 300, 200
```

## MSGBOX Statement

Used to pop up a simple message box with a message and an OK button.

### Syntax

```
MSGBOX string
```

### Remarks

This statement pops up a simple message box with an OK button and waits for the user to press the OK button before continuing.

### See also

DIALOG, DIALOGBOX

### Example

This example pops up a simple message box on the screen.

```
msgbox "Hello world!"
```

## NAME Statement

Renames or moves a file.

### Syntax

```
NAME oldfilespec AS newfilespec
```

### Remarks

*oldfilespec* is the current file name.

*newfilespec* is the new file name.

This command works in a similar way to the DOS RENAME command, with the added ability to move a file, if the new filename includes a path.

You cannot use the NAME command to rename a directory.

If an error occurs during the rename operation, the ERR_FILERENAME error will be generated.

### See also

DEL, FINDFIRST, FINDNEXT, KILL

### Example

This example renames the file "test.fil" to the file "test.dat"

```
name "test.fil" as "test.dat"
```

## NEXT Statement

Marks the end of a FOR ... NEXT loop. See the FOR command for more information and examples.

## NOT Operator

Performs a bitwise logical NOT operation.

### Syntax

```
NOT op
```

### Remarks

This operator takes the binary representation of *op* and reverses the status of each bit in the representation.

When using NOT in an IF statement, the value FALSE (zero) will be converted to TRUE (–1) and vice versa.

### See also

AND, EQV, IF, IMP, OR, XOR

### Example

This example prints out the result of using NOT on various values.

```
print not true
print not false
print not 1
```

## OCT Function

Converts an integer expression to an octal (base 8) representation.

### Syntax

```
OCT(expr)
```

### Remarks

The expression passed to this function will be converted to a long integer before being converted to an octal representation.

### See also

HEX, STR, VAL

### Example

This example prints the octal representation of the number 42.

```
print oct(42)
```

## OPEN Statement

Opens a file so the program can perform input and output operations.

### Syntax

```
OPEN file FOR accessmode AS [#]filenum [LEN = reclen]
```

### Remarks

*file* is the name of the file to open.

*accessmode* specifies the way the program will write to or read from file: INPUT, OUTPUT, APPEND, RANDOM and BINARY.

*filenum* is the file number to assign to the file.

*reclen* is the length of a record in a sequential or random-access file, in the range 1 through 32767.

### See also

CLOSE, FREEFILE

### Example

This example opens a file for sequential output and writes a string of test data to it.

```
open "test.dat" for output as #1
print #1, "this is a test"
close #1
```

## OR Operator

OR performs a bitwise logical or operation between its operands.

### Syntax

```
op1 OR op2
```

### Remarks

*op1* and *op2* are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the OR operator:

| a | b | a OR b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

### See also

AND, EQV, IMP, NOT, XOR

### Example

```
print "5 or 6 is 7: "; 5 or 6
```

## PAUSE Statement

Used to suspend script execution for a certain time interval.

### Syntax

```
PAUSE time
```

### Remarks

time is the amount of time to suspend execution, expressed in seconds. If you want to delay for less than a second, use ordinary decimal notation.

This command is identical to the script DELAY command.

### See also

DELAY, WAITFOR, WHEN QUIET, WHEN TIME

### Example

This example sends "atz" to initialize the modem, pauses for half a second, and sends a command to dial the modem.

```
send "atz"
pause 0.5
send "atdt5551212"
```

## PLAY Statement

Used to play a .wav file using the Windows multimedia services.

### Syntax

```
PLAY filename
```

### Remarks

If Windows cannot find the named .wav file to play, it will sound a default beep (which may be another .wav file). The .wav file is played in "asynchronous" mode which means that Windows will not wait for the sound to finish, and will continue execution with the next script command immediately.

If you would like more control over the options used to play the sound, use the following declaration:

```
declare sub sndPlaySound lib "mmsystem" (fname as string, flags as
integer)
```

This declaration declares a new subroutine, sndPlaySound, which actually calls the Windows multimedia sound player directly. For more information on the flags parameter, see the Windows SDK documentation.

### See also

BEEP, SOUND

### Example

This example plays the sound in the file "soundfil.wav".

```
PLAY "soundfil.wav"
```

## POS Function

Reports the current horizontal coordinate position (column number) of the cursor.

### Syntax

```
POS
```

### Remarks

The return value of this function ranges from 1 to the maximum number of columns on the screen (either 80 or 132).

### See also

CSRLIN, LOCATE

### Example

This example prints the current horizontal cursor position.

```
print pos
```

## PRINT Statement

Outputs data to the display, communications port, or a file.

### Syntax

```
PRINT [#filenum, ][expression[[;|,]expression ...]][;|,]
```

### Remarks

*expression* is an expression of any type except a user defined type.

Multiple items can be separated with semicolons (to display the next item immediately after the previous item), or commas (to display the next item in the next 14-character wide "print zone"). A semicolon at the end of the line will suppress the automatic carriage return and line feed that occurs after printing.

The PRINT statement by itself moves the cursor to the beginning of the next line.

### See also

LOCATE, TAB

### Example

This example shows the famous "Hello world" program in its entirety.

```
print "Hello, world!"
```

## PRINTER Statement

Turns the printer toggle on or off.

### Syntax

```
PRINTER onoff
```

### Remarks

This command turns the printer toggle on and off. When on, all incoming data is echoed to the printer as well as the screen. The destination of the printer data is controlled by the "Print file" option in Options/Files/File definitions.

### See also

CAPTURE

### Example

This example shows how to turn the printer on, wait for some specified data from the port, then turn the printer off again.

```
print on
send
waitfor "Ready"
print off
```

## PUT Statement

Writes data to a random access or binary file, from record variables defined by a TYPE ... END TYPE statement.

### Syntax

```
PUT [#]filenumber, [position], variable
```

### Remarks

*filenumber* is the file number assigned to the open file.

*position* is the number of the random access file record, or binary file byte to begin writing. Note that unlike some other languages, the first record or byte in the file is number 1, not number 0. If this position is not specified, then the record is written to the current file position.

*variable* is the name of the variable that contains the data to place in the file.

### See also

GET, INPUT, OPEN, TYPE

## Example

This example creates a random access file, writes a record to it from the variable d, and reads it back into the variable z. It then prints out the contents of z to make sure that the operation succeeded.

```
type daterec
  day as integer
  month as integer
  year as integer
end type
dim d as daterec, z as daterec
d.day = 11
d.month = 10
d.year = 1993
open "test.dat" for random as #1 len = len(daterec)
put #1, 1, d
get #1, 1, z
close #1
print z.day, z.month, z.year
```

## REAL Type

Used to declare a variable that can handle real (floating-point) numbers.

### Remarks

The range of REAL numbers is -3.4e38 to 3.4e38.

### See also

BYTE, DIM, INTEGER, LONG, STRING, TYPE

### Example

This example declares a variable of type real, assigns a value to it, and prints out the value.

```
dim x as real
x = 1.5
print x
```

## RECEIVE Statement

Captures a line of data from the communications port (by default) and places the data into a string variable.

### Syntax

```
RECEIVE [#filenum,] var
```

### Remarks

This statement is identical to the INPUT statement except that if a file number is not specified, the data is input from the communications port instead of the keyboard.

### See also

INPUT

### Example

This example reads a line of input from the communications port.

```
dim s as string
receive s
```

## RECEIVEFAX Statement

Used to receive incoming faxes.

### Syntax

```
RECEIVEFAX [CLASS]
```

### Remarks

The class parameter specifies the fax class to answer. Currently supported are 1 and 2. This should only be called with a specific class AFTER the modem has reported a fax connect.

If the class parameter is omitted, this will receive a fax assuming the phone is ringing. The fax class will be automatically detected).

### See Also

SENDFAX

### Example

```
RECEIVEFAX 1
```

## RECEIVEFILE Function

Used to receive a file or files from a remote computer.

### Syntax

```
RECEIVEFILE(filename, protocol)
```

### Remarks

This function initiates a file transfer to receive files from a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

*protocol* is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT, BPLUS

For the first five protocols, the *filename* parameter must specify an actual file name in which to place the received file. The ASCII and Xmodem variant protocols do not supply a filename, so one must be supplied in the DOWNLOAD function.

For the last five protocols, the *filename* parameter should be the name of a directory in which to place the received files. If this parameter is an empty string ("") then the download directory specified in **Options/Files/Path Definitions** will be used.

The RECEIVEFILE function returns zero if the file transfer was successful. If the transfer was unsuccessful, RECEIVEFILE returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the DOWNLOAD function.

### See also

DOWNLOAD, SENDFILE, UPLOAD

## Example

This example receives a file using the Zmodem protocol, assuming the remote computer has already started the transfer.

```
if receivefile("", Zmodem) = 0 then
  print "file transfer ok!"
end if
```

## REM Statement

Allows you to include a comment, or non-executing text for your own reference, in a program.

### Syntax

```
REM comment
or
' comment
```

### Remarks

The apostrophe ' character is a synonym for the REM statement.

Either variation of the statement will cause the compiler to ignore all text on a line following the ' or REM statement.

When adding a comment to a line of executable source code, the REM must be preceded by a colon (:). A colon is not required if you are using the apostrophe.

Comments do not affect the size or execution speed of the compiled script.

### Example

This example shows two comment lines and one line that is not a comment.

```
rem This is a comment.
' This is another comment, like the above line.
print "This is not a comment."
```

## RESET Statement

### Description

Closes all open files.

### Syntax

```
RESET
```

### Remarks

The RESET statement works the same way as the CLOSE statement without parameters. The functions CLOSE, END, STOP, and SYSTEM also close any open files.

### See also

CLOSE, END, STOP, SYSTEM

### Example

This example opens a file for input, then closes all open files.

```
open "test.dat" for input as #1
reset
```

## RESETEMULATION Statement

Resets the current emulation.

### Syntax

```
RESETEMULATION
```

### Remarks

This command resets the operation of the currently selected emulation. It is identical to the Reset Emulation command on the Terminal menu.

### See also

EMULATION, RESETMODEM

### Example

This example shows how the RESETEMULATION command might be used.

```
hangup
resetemulation
```

## RESETMODEM Statement

Sends the modem reset string as specified in the modem configuration.

### Syntax

```
RESETMODEM
```

### Remarks

This command sends the "Modem init" string as specified in the modem configuration.

### See also

AUTOANSWER, RESETEMULATION

### Example

This example shows how the RESETMODEM command might be used to reset the modem after hanging up.

```
hangup
resetmodem
```

## RETURN Statement

Returns from a subroutine started with the GOSUB statement. See the GOSUB statement for more information and examples.

## RIGHT Function

Returns a string consisting of a specified number of characters starting at the right (end) of a string.

### Syntax

```
RIGHT(string, num)
```

### Remarks

*string* is any string expression.

*num* is any number in the range 0 through 32767.

If there are fewer than num characters in the string, the entire string is returned even though it is shorter than num.

### See also

INSTR, LEFT, MID

### Example

This example shows how the RIGHT function can be used to extract the rightmost characters from another string.

```
dim a as string
a = "QmodemPro for Windows"
print right(a, 7)
```

## RMDIR Statement

Removes a directory from a disk.

### Syntax

```
RMDIR directory
```

### Remarks

*directory* is the name of the directory to remove.

You cannot remove a directory if it still contains files.

If the directory cannot be removed, the ERR_PATH error is generated. Use the CATCH statement to respond to this error.

### See also

CHDIR, KILL, MKDIR

### Example

This example removes the directory "c:\qmwin\test", assuming it is empty.

```
rmdir "c:\qmwin\test"
```

## RND Function

Returns a pseudo-random real number between greater than or equal to 0 and less than 1.

### Syntax

```
RND[(number)]
```

### Remarks

If *number* is omitted or greater than zero, the next random number in sequence is generated.

If *number* is less than zero, RND returns the same random number every time, depending on the value of expr.

If *number* is zero, the last random number generated is returned.

### Example

This example shows the three ways of using the RND function.

```
print rnd(-0.5)
print rnd
print rnd(0)
```

## RTRIM Function

Trims trailing spaces (spaces on the right) from a string expression.

### Syntax

```
RTRIM(string)
```

### Remarks

*string* is any string expression. Any trailing spaces are removed from the string and the result is returned.

### See also

LTRIM

### Example

This example shows how the RTRIM function can be used to trim trailing spaces from a string.

```
dim a as string
a = "  QmodemPro for Windows  "
print ">"; rtrim(a); "<"
```

## SCREEN Function

Returns the character or color attribute at a particular screen position.

### Syntax

```
SCREEN(x, y [, attr])
```

### Remarks

Returns the character or attribute (color) at a particular screen position. The screen position is given by the *x* and *y* parameters and ranges from 1 to the with or height of the terminal screen.

If the *attr* parameter is omitted or is zero, this function returns the ASCII value of the character on the screen at the given position. If the attr parameter is one, the return value is the attribute of the character on the screen at the given position.

The attribute returned is defined by the following formula:

```
b * 16 + f
```

where b is the background color and f is the foreground color, as listed under the COLOR statement.

### See also

COLOR

### Example

This example clears the screen, writes "Hi" in gray on blue, and prints the ASCII value of "H" followed by the color value 71.

```
cls
color 7, 4
print "Hi"
print screen(1, 1), screen(2, 1, 1)
```

## SCROLLBACK Statement

Turns scrollback mode on or off.

### Syntax

```
SCROLLBACK onoff
```

### Remarks

This command turns the scrollback mode on or off. The operation of the script is not affected while in scrollback mode (ie. the script continues to execute).

### See also

CAPTURE, SCROLLBACKRECORD

### Example

This example shows how to capture some data, then automatically go to scrollback mode to review it.

```
send
waitfor "Command"
scrollback on
```

## SCROLLBACKRECORD Function

Used to get or set the current state of the scrollback record toggle.

### Syntax

```
SCROLLBACKRECORD
or
SCROLLBACKRECORD(onoff)
```

### Remarks

There are two forms to the SCROLLBACKRECORD function. The first form takes no arguments and simply returns the current setting. The second form sets the state of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means scrollback record mode is turned on. A zero value means scrollback record mode is turned off.

When scrollback record mode is on, incoming data is captured to the scrollback buffer for later review. When scrollback record mode is turned off, data is not copied to the scrollback buffer.

### See also

CAPTURE, SCROLLBACK

### Example

This example shows how the SCROLLBACKRECORD function might be used to automatically capture some data.

```
scrollbackrecord on
send
waitfor "End of data"
scrollbackrecord off
```

## SEEK Statement

Sets the current position in an open file for the next I/O operation.

### Syntax

```
SEEK [#]filenumber, position
```

### Remarks

*filenumber* is the number allocated to the open file.

*position* is the number of the record or byte to be accessed. Note that unlike many programming languages, the records and bytes in a file are numbered starting at 1 rather than at 0.

### See also

GET, LOC, OPEN, PUT

### Example

This example reads from "c:\autoexec.bat" the string starting at offset 50.

```
dim s as string
open "c:\autoexec.bat" for input as #1
seek #1, 50
input #1, s
close #1
```

## SELECT CASE Statement

Allows a number of expressions to be compared, executing statements based on the results of the comparison.

### Syntax

```
SELECT CASE expression
  CASE case-condition[, case-condition ...]
    [statements]
  [CASE case-condition[, case-condition ...]
    [statements]]
  [CASE ELSE
    [statements]]
END SELECT
```

### Remarks

*expression* is a numeric or string expression.

*case-condition* is a condition in one of the following forms:

```
expression
expression TO expression
IS relational-operator expression
```

If the *expression* matches the *case-condition*, the statements matching the corresponding CASE are executed. Control then passes to the first statement following END SELECT.

If none of the *case-conditions* matches and there is a CASE ELSE clause, the statements within the CASE ELSE clause are executed.

### See also

IF

## Example

This example tests the numbers 0 through 11 with a SELECT CASE statement and prints the results.

```
dim i as integer
for i = 0 to 11
  select case i
    case 1
      print "i is 1"
    case 2 to 9
      print "i is between 2 and 9 inclusive"
    case is >= 10
      print "i is greater than or equal to 10"
    case else
      print "i must be zero or negative"
  end select
next i
```

## SEND Statement

Outputs data to the communications port.

### Syntax

```
SEND [expression[[;|,]expression ...]][;|,]
```

### Remarks

*expression* is an expression of any type except a user defined type.

Multiple items can be separated with semicolons (to display the next item immediately after the previous item), or commas (to display the next item in the next 14-character wide "print zone").

A semicolon at the end of the line will suppress the automatic carriage return after the transmitted data.

The SEND statement by itself sends only a carriage return.

### See also

LASTCONNECTPASSWORD, LASTCONNECTUSERID, PRINT

### Example

This example shows how the SEND command may be used to send various data to the modem.

```
send "bob"          ' sends "bob" followed by Enter
send "g";           ' sends "g" without an Enter
send lastconnectuserid
                    ' sends the UserId corresponding
                    ' to the phone directory entry
                    ' you are currently connected to
```

## SENDFAX Statement

Used to send faxes to a remote computer.

### Syntax

```
SENDFAX number, to, from, subject, fine, cover, file, font, fontsize
```

### Remarks

The fields **number, to, from, subject, file** and **font** are strings. The fields **fine** and **cover** are True or False. **Fontsize** is an integer.

**Number** is the phone number to send the fax to. **To** and **From** are the names of the persons sending and receiving the fax. **Subject** is the subject of the fax. **File** is the name of the file to send (TXT, PCX or BMP), **Font** is the name of the Windows font to use to rasterize the text, **Fontsize** is the size of the font in points.

### See Also

RECEIVEFAX

### Example

```
SENDFAX "805-873-2599", "Sales", "John", "Upgrade", FALSE, FALSE,
"PURCHORD.TXT", "Terminal", 10
```

## SENDFILE Function

Used to send files to a remote computer.

### Syntax

```
SENDFILE(filename, protocol)
```

### Remarks

This function initiates a file transfer to send files to a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

The protocol is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT, BPLUS

The first five protocols require a single file name in the filename parameter. This file name indicates the file to send.

The last five protocols can accept more than just one file in the filename parameter — separate multiple filenames with spaces.

The SENDFILE function returns zero if the file transfer was successful. If the transfer was unsuccessful, SENDFILE returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the UPLOAD function.

### See also

DOWNLOAD, RECEIVEFILE, UPLOAD

## Example

This example shows how to send a file using the Zmodem protocol, assuming the remote computer has been told to accept a file transfer.

```
if sendfile("c:\qmwin\test.dat", Zmodem) = 0 then
  print "file transfer ok!"
end if
```

## SETCOMM Statement

Sets the communications parameters without changing the active device.

### Syntax

```
SETCOMM commsettings
```

### Remarks

The *commsettings* parameter should be a string in the following format:

```
"baudrate,DPS"
```

*baudrate* is the baud rate.

*D* is the number of data bits.

*P* is the type of parity: N for None, E for Even, O for Odd, M for Mark, or S for Space.

*S* is the number of stop bits.

### See also

DEVICE

### Example

This example sets the communications port to 19200 baud, 8 data bits, no parity bit, and one stop bit.

```
setcomm "19200,8N1"
```

## SETDTR Statement

Sets the DTR (Data Terminal Ready) signal high (ON) or low (OFF).

### Syntax

```
SETDTR on | off
```

### Remarks

Most modems will disconnect when the DTR signal is lowered. This may be more reliable than using the "hangup" command.

### See also

HANGUP

### Example

This example sends a command to log off an on-line service, waits five seconds, then hangs up the modem.

```
send "bye"
delay 5
setdtr off
```

## SGN Function

Returns an integer indicating the sign of a number.

### Syntax

```
SGN(expr)
```

### Remarks

If expr is greater than zero, the return value is 1. If expr is less than zero, the return value is −1. If expr is equal to zero, the return value is zero.

### See also

ABS

### Example

This example shows what happens to a negative number, zero, and a positive number with the SGN function.

```
print sgn(-5), sgn(0), sgn(3)
```

## SHELL Statement

Runs a DOS command or other executable program or batch file t
script.

### Syntax

```
SHELL [command]
```

### Remarks

*command* is a string expression containing an executable DOS or Windows command.

### See also

END, SYSTEM

### Example

This example runs Windows Control Panel.

```
shell "control"
```

## SIN Function

Returns the sine of an angle.

### Syntax

```
SIN(angle)
```

### Remarks

angle is the measurement of an angle expressed in radians. You can compute radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

### See also

ATN, COS, TAN

### Example

This example prints the sine of 1 radian.

```
print sin(1)
```

## SIZE Statement

Used to resize the *QmodemPro for Windows* application window on the Windows desktop.

### Syntax

```
SIZE width, height
```

### Remarks

width is the new width of the window on the desktop. Width values range from 0 to the width of your screen.

Height is the new height of the window on the desktop. Height values range from 0 to the height of your screen.

If you are using standard VGA with a 640x480 screen then the width values can range from 0 through 640 and height values can range from 0 through 480.

### See also

ACTIVATE, MAXIMIZE, MINIMIZE, MOVE

### Example

This example changes the size of the *QmodemPro for Windows* application window to be 400 pixels wide and 300 pixels high.

```
SIZE 400, 300
```

## SOUND Statement

Generates a tone of a specific frequency and duration.

### Syntax

```
SOUND frequency, duration
```

### Remarks

This statement generates a tone of a specific frequency and duration. The frequency is specified in hertz (cycles per second) and the duration is specified in seconds.

### See also

BEEP, PLAY

### Example

This example sounds a 1000 Hz tone for 2/10 of a second.

```
SOUND 1000, 0.2
```

## SPACE Function

Returns a string consisting of a specified number of spaces.

### Syntax

```
SPACE(number)
```

### Remarks

*number* is the number of spaces to return.

This function is identical to the SPC function.

### See also

SPC, TAB

### Example

This example prints two words separated by 20 spaces.

```
print "here"; space(20); "there"
```

## SPC Function

Returns a string consisting of a specified number of spaces.

### Syntax

```
SPC(number)
```

### Remarks

*number* is the number of spaces to return.

This function is identical to the SPACE function.

### See also

SPACE, TAB

### Example

This example prints two phrases separated by 30 spaces.

```
print "neither here"; spc(30); "nor there"
```

## SPLITSCREEN Statement

Turns split screen mode on or off.

### Syntax

```
SPLITSCREEN onoff
```

### Remarks

This command turns the split screen mode on or off. The operation of the script is not affected while in split screen mode (ie. the script continues to execute).

### See also

SCROLLBACK

### Example

This example shows how to turn on and off the split screen mode from a script.

```
splitscreen on
splitscreen off
```

## SQR Function

Returns the square root of a numeric expression.

### Syntax

```
SQR(expression)
```

### Remarks

*expression* must evaluate to a number greater than or equal to zero.

If the expression is less than zero, SQR will generate the ERR_MATH error. This error can be caught with the CATCH statement.

### See also

EXP, LOG

### Example

This example prints the square root of 9.

```
print sqr(9)
```

## STAMP Statement

Used to write a string of text to the log file, if it is open.

### Syntax

```
STAMP string
```

### Remarks

*string* is a string that is written to the log file. If the log file is not currently open, this statement has no effect.

### See also

CAPTURE, LOGFILE

### Example

This example opens the log file, writes a stamp string to it, and closes the log file.

```
logfile on
stamp "This is a stamped string."
logfile off
```

## STATIC Statement

Used to declare variables in a subroutine or function that retain their values even after the subroutine or function has returned to its caller. The variables can be used when the subroutine or function is called again without being reinitialized to zero or a blank string.

### Syntax

```
STATIC var[([lowerbound TO] upperbound)] [AS type][, var[([lowerbound
TO] upperbound)] [AS type]...]
```

### Remarks

The syntax of the STATIC statement is identical to that of the DIM statement. For a complete discussion of variable declarations please see the DIM statement.

### See also

DIM

### Example

This example shows how a STATIC variable might be used to count the number of times a function has been called.

```
declare sub test
call test
call test

sub test
  static i as integer
  i = i + 1
  print "test has been called "; i; " times."
end sub
```

## STOP Statement

Terminates execution of a script.

### Syntax

```
STOP
```

### Remarks

The STOP statement is similar to the END statement except that it puts a warning message on the screen after the script has terminated.

### See also

END

### Example

This example uses STOP to terminate a loop.

```
dim i as integer
i = 0
do
  i = i + 1
  if i > 99 then stop
loop
```

## STR Function

Converts a number to a string representation of the number.

### Syntax

```
STR(number)
```

### Remarks

This function returns the string representation of *number*.

This function is the opposite of the VAL function, which converts a number in string format to an integer.

### See also

VAL

### Example

This example uses the STR function to convert a number to a string and count the number of digits in the number.

```
dim a as string
a = str(47)
print a; "is "; len(a); " digits long."
```

## STRING Type

Used to declare a variable that can handle variable length character data, or to define a string that can hold a specific number of characters.

### Remarks

Variables of string type can hold values that are up to 32767 characters in length.

To declare a fixed length string that holds only a specific number of characters, use the syntax

```
STRING*n
```

where n is the number of characters you want the string to hold. Fixed length strings are useful because variable length strings are not allowed within user defined type declarations.

### See also

DIM, TYPE

### Example

This example declares a string, assigns a value to it, and prints the value.

```
dim a as string
a = "Hello, world!"
print a
```

## STRING Function

Repeats a character a specified number of times, and returns the results as a string.

### Syntax

```
STRING(number, code)
or
STRING(number, string)
```

### Remarks

*number* is the number of characters to output, in the range 0 to 32767.

*code* is the ASCII code for the character you want, in the range 0 to 255.

In the second syntax, *string* is a string and the STRING function returns the first character of *string* repeated *num* times.

### See also

CHR, SPACE

### Example

This example prints a string of 50 dashes on the screen.

```
print string(50, "-")
```

## STRIPHIBIT Function

Used to get or set the current state of the "8th Bit Strip" toggle.

### Syntax

```
STRIPHIBIT
or
STRIPHIBIT(onoff)
```

### Remarks

There are two forms to the STRIPHIBIT function. The first form takes no arguments and simply returns the current setting. The second form sets the state of the toggle and returns the previous state.

In both the parameter and the return value, a nonzero value means the high bit is stripped from incoming characters. A zero value means the high bit is not stripped.

### See also

DUPLEX

### Example

This example shows how the STRIPHIBIT function can be used in a logon script to turn on the high bit strip.

```
dim oldstrip as integer
oldstrip = striphibit(on)
waitfor "UserID"
send "123456"
striphibit oldstrip
```

## SUB Statement

Allows you to define your own subprograms that do not directly return a value to the caller.

### Syntax

```
SUB name[(arg AS type[, arg AS type]...)]
  ...
END SUB
```

### Remarks

*name* is the name you assign to the subroutine.

*arg* is the name of a formal argument to the subroutine. Each argument must have an associated type declaration using the AS keyword.

You may define local variables within your user-defined subroutine. You can use STATIC declarations to preserve the value of individual variables across subroutine calls.

You can execute a sub-program using the CALL statement, along with any arguments you wish to pass to the sub-program.

### See also

CALL, DECLARE, EXIT, FUNCTION, STATIC

### Example

This example defines a simple subroutine and shows two ways of calling it.

```
sub test(i as integer)
  print "in subroutine test: "; i
end sub

sub 1
call sub(5)
```

## SYSTEM Statement

Closes all open files and terminates a script.

### Syntax

```
SYSTEM
```

### Remarks

This statement immediately ends the script and closes the *QmodemPro for Windows* application.

### See also

END, SHELL, STOP

### Example

This example terminates *QmodemPro for Windows* after waiting for a key to be pressed.

```
print "press a key to exit...";
while inkey = ""
wend
system
```

## TAB Function

Returns the appropriate number of spaces to move the cursor to the specified column on the screen.

### Syntax

```
TAB(column)
```

### Remarks

This function returns the appropriate number of spaces to move the cursor to a particular column on the screen. Note that this function only works on the actual terminal screen.

### See also

SPC

### Example

This example shows how the TAB function might be used to print columnar data on the screen.

```
print "test"; tab(40); "column 40"
```

## TAN Function

Returns the tangent of an angle.

### Syntax

```
TAN(angle)
```

### Remarks

angle is the measurement of an angle expressed in radians. You can compute radians to degrees by multiplying by 180/pi (pi is approximately 3.14159).

### See also

ATN, COS, SIN

### Example

This example prints the tangent of 1 radian.

```
print tan(1)
```

## TIME Function

Returns the current date from the computer's internal clock.

### Syntax

```
TIME
```

### Remarks

The TIME function returns the current time in the form specified in Windows Control Panel, International section.

### See also

DATE

### Example

This example prints the current time according to Windows.

```
print "The time is now "; time
```

## TIMEOUT Function

Used to either set or retrieve the current script timeout setting.

### Syntax

```
TIMEOUT
or
TIMEOUT(seconds)
```

### Remarks

When TIMEOUT is called without an argument, it returns the current script timeout value.

When TIMEOUT is called with a *seconds* argument, it sets the current timeout value to the specified number of seconds. In addition, it returns the previous timeout value.

If the timeout is set to zero, the timeout function is disabled.

The script timeout value applies to INPUT, RECEIVE, and WAITFOR commands.

### See also

INPUT, RECEIVE, WAITFOR, WHEN

### Example

This example shows how you might modify the timeout value during a logon script.

```
timeout 20
waitfor "what is your first name"
send "joe"
...
catch err_timeout
hangup
end
```

## TIMER Function

Returns the number of seconds since midnight.

### Syntax

```
TIMER
```

### Remarks

TIMER returns the number of seconds since midnight as a real type value.

### See also

TIMEOUT, WAITFOR, WHEN TIME

### Example

This example prints the number of seconds since midnight, according to Windows.

```
print timer
```

## TRAPSCREEN Statement

Used to capture the current terminal screen to a file.

### Syntax

```
TRAPSCREEN filename
```

### Remarks

This function takes a snapshot of the current terminal screen and appends it to the named file. A date and time stamp precedes the data on the screen.

### See also

CAPTURE, SCROLLBACK

### Example

This example shows how the TRAPSCREEN command might be used to capture some data from the remote host.

```
send
waitfor "OK"
trapscreen "snapshot.txt"
```

## TYPE ... END TYPE Statement

Allows you to create a data structure (data type or user-defined type) consisting of one or more fields.

### Syntax

```
TYPE typename
  field AS typename
  ...
END TYPE
```

### Remarks

*typename* is the name of the new type.

*fieldname* is an element of the user-defined data type.

*typename* is the type of the field. It can be of type BYTE, INTEGER, LONG, REAL, or STRING*n (variable length strings are not permitted in user-defined types), or another user-defined type.

Variables of your new user-defined type are declared just like regular variables using the DIM statement. To access a field of your user defined variable, use a period after the variable name followed by the field name. See the example for an illustration of this.

### See also

DIALOG, DIM

## Example

This example declares a user-defined type called "daterec", declares a variable d of the type, assigns values to each of its fields, and prints the values of each of its fields.

```
type daterec
  day as integer
  month as integer
  year as integer
end type
dim d as daterec
d.day = 11
d.month = 10
d.year = 1993
print d.day, d.month, d.year
```

## UCASE Function

Returns a copy of a string with all lower case characters converted to upper case.

### Syntax

```
UCASE(string)
```

### Remarks

*string* can be any string expression.

This operation is useful for making case insensitive comparisons of text. The LCASE function operates similarly, but converts the specified text to lower case.

### See also

LCASE

### Example

This example shows how the UCASE function converts all lower case characters in a string to upper case.

```
dim a as string
a = "QmodemPro for Windows"
print ucase(a)
```

## UPLOAD Function

Used to send files to a remote computer.

### Syntax

```
UPLOAD(filename, protocol)
```

### Remarks

This function initiates a file transfer to send files to a remote computer. At the time this command is executed, the remote computer must already have started the file transfer. If you are connecting to a bulletin board system (BBS) then your script should already have sent the command to the BBS that will start the file transfer.

*protocol* is one of the following predefined constants:

ASCII, XMODEM, XMODEMCRC, XMODEM1K, XMODEM1KG, YMODEM, YMODEMG, ZMODEM, KERMIT, BPLUS

The first five protocols require a single file name in the *filename* parameter. This file name indicates the file to send.

The last five protocols can accept more than just one file in the *filename* parameter — separate multiple filenames with spaces.

The UPLOAD function returns zero if the file transfer was successful. If the transfer was unsuccessful, UPLOAD returns the error code describing the error. For a list of error codes see Chapter 4.

This function is identical to the SENDFILE function.

### See also

DOWNLOAD, RECEIVEFILE, SENDFILE

## Example

This example starts a file transfer of "c:\qmwin\test.dat" to a remote computer. It assumes that the remote computer is prepared to receive the file.

```
if upload("c:\qmwin\test.dat", Zmodem) = 0 then
  print "file transfer ok!"
end if
```

## VAL Function

Returns the numeric value of a string.

### Syntax

```
VAL(string)
```

### Remarks

The VAL function begins evaluating a string at the first character, and scans it until the end of the string, or until a non-numeric character is reached, whichever comes first. It returns the value of the number found in the string, if any (if no number is found, zero is returned).

The VAL function does not support REAL numbers.

### See also

STR

### Example

This example converts the string containing the digits 4 and 2 to the decimal value 42.

```
dim a as string, i as integer
a = "42"
i = val(a)
print i
```

## VERSION Function

Returns the *QmodemPro for Windows* version number as a string.

### Syntax

```
VERSION
```

### Remarks

In version 1.10 of *QmodemPro for Windows*, this function returns the string "1.10". You can always assume that the version numbers will be increasing with respect to the string comparison operators.

### See also

DOSVERSION, WINVERSION

### Example

This example prints the current *QmodemPro for Window* version number.

```
print "QmodemPro for Windows "; version
```

## VIEWFILE Statement

Used to invoke the internal file viewer with a specified file.

### Syntax

```
VIEWFILE filename
```

### Remarks

This function brings up the internal file viewer with the named file loaded ready for viewing. Note that that script continues to run after the viewer is started.

### See also

EDITFILE, VIEWGIF

### Example

This example shows how the VIEWFILE command might be used to view the host mode user file.

```
viewfile "host.usr"
```

## VIEWGIF Statement

Used to invoke the internal GIF viewer with a specified file.

### Syntax

```
VIEWGIF filename
```

### Remarks

This function brings up the internal GIF viewer with the named GIF or BMP file loaded ready for viewing. Note that that script continues to run after the viewer is started.

### See also

EDITFILE, VIEWFILE

### Example

This example shows how the VIEWGIF command might be used to view a GIF picture.

```
viewgif "fastcar.gif"
```

## WAITFOR Statement

Used to wait for a particular sequence of characters to appear from the communications port.

### Syntax

```
WAITFOR string [, timeout]
```

### Remarks

WAITFOR suspends script execution until the specified string appears from the communications port. By default, the timeout value is whatever was last set by the TIMEOUT command. If a timeout value is specified, it is used instead of the current TIMEOUT value.

The case of letters is ignored when comparing incoming data against the specified string. However, terminal emulation escape sequences are not ignored.

If the character string does not appear within the timeout period, the ERR_TIMEOUT error is generated. This can be caught with the CATCH script statement to take corrective action.

### See also

RECEIVE, WHEN

### Example

This example shows how the WAITFOR command might be used in a script to log on to an on-line service.

```
waitfor "first name"
send "john doe"
waitfor "password"
send "sesame"
```

## WEND Statement

Used to mark the end of a WHILE ... WEND statement. See the WHILE statement for more information and examples.

## WHEN CLEAR Statement

Used to remove all when conditions, a set of when conditions, or a particular when condition.

### Syntax

```
WHEN CLEAR ALL
or
WHEN CLEAR name
```

### Remarks

WHEN CLEAR ALL clears all the currently active WHEN conditions.

WHEN CLEAR name clears all the WHEN conditions that have the name *name*.

After a WHEN condition is cleared, it is no longer active.

### See also

WHEN DISABLE, WHEN ENABLE, WHEN

### Example

This example shows how a WHEN CLEAR command might be used to remove a WHEN MATCH condition that is no longer needed.

```
when match "press enter" do send
waitfor "main menu"
when clear all
```

## WHEN DISABLE Statement

Used to disable all when conditions, a set of when conditions, or a particular when condition.

### Syntax

```
WHEN DISABLE ALL
or
WHEN DISABLE name
```

### Remarks

WHEN DISABLE ALL disables all the currently active WHEN conditions.

WHEN DISABLE name disables all the WHEN conditions that have the name *name*.

A disabled WHEN condition can be reactivated with the WHEN ENABLE statement.

### See also

WHEN CLEAR, WHEN ENABLE, WHEN

### Example

This example shows how the when condition with the name "Enter" might be disabled if it is temporarily unnecessary.

```
when name "Enter" match "press enter" do send
...
when disable "Enter"
```

## WHEN ENABLE Statement

Used to enable all when conditions, a set of when conditions, or a particular when condition.

### Syntax

```
WHEN ENABLE ALL
or
WHEN ENABLE name
```

### Remarks

WHEN ENABLE ALL enables all the currently active WHEN conditions.

WHEN ENABLE name enables all the WHEN conditions that have the name *name*.

### See also

WHEN CLEAR, WHEN DISABLE, WHEN ENABLE, WHEN

### Example

This example shows how the WHEN ENABLE command might be used to turn on a WHEN MATCH event that has been previously disabled.

```
when name "Enter" match "press enter" do send
...
when disable "Enter"
...
when enable "Enter"
```

## WHEN Statement

Used set up a mechanism that will trigger a particular statement or set of statements when a certain event happens.

### Syntax

```
WHEN [NAME name] {MATCH string | QUIET seconds | TIME time} DO
statements

or

WHEN [NAME name] {MATCH string | QUIET seconds | TIME time} DO
  statements ...
END WHEN
```

### Remarks

There are three kinds of WHEN trigger events: MATCH, QUIET, and TIME.

A MATCH event watches the communications port for a particular sequence of characters and executes the statements associated with the WHEN statement when the sequence of characters is seen. Upper and lower case characters are considered the same when comparing characters.

A QUIET event watches the activity on the communications port and executes the statements associated with the WHEN statement when there is no activity for a certain amount of time. This is often used to send an Enter whenever there is no data received from the remote computer for, say, 10 seconds.

A TIME event watches the time and executes the statements associated with the WHEN statement when a particular time of day occurs. The time of day is specified in string format, so "2:00" would be 2:00am. Hours, minutes, and seconds are accepted separated by colons. An AM or PM indicator is also accepted.

## See also

WAITFOR, WHEN CLEAR, WHEN DISABLE, WHEN ENABLE

## Example

This example shows how the WHEN MATCH command might be used to skip past logon screens when waiting for a "main menu" prompt from an on-line service.

```
when match "press enter" do send
waitfor "main menu"
```

## WHILE ... WEND Statement

Repeats a series of statements while a given condition is true.

### Syntax

```
WHILE expression
  [statements]
WEND
```

### Remarks

*expression* is any numeric expression or variable that evaluates to a nonzero value (true) or zero (false). A WHILE ... WEND loop will execute so long as the expression is true, then when the expression becomes false, then passes control to the statement that follows the WEND.

The WHILE ... WEND provides an alternate syntax to the DO WHILE ... LOOP statement.

Each WHILE must have a corresponding WEND. Unmatched statements will cause an error message.

### See also

DO ... LOOP, FOR ... NEXT

### Example

This example shows how a WHILE loop might be used to print out the first 10 whole numbers.

```
dim i as integer
i = 0
while i < 10
  print "i is "; i
  i = i + 1
wend
```

## WINVERSION Function

Returns the current Windows version as a string.

### Syntax

```
WINVERSION
```

### Remarks

This function returns the Windows version as a string in the format X.Y. X is the major version number and Y is the minor revision number.

For example, Windows version 3.1 would be reported as "3.10".

### See also

DOSVERSION, VERSION

### Example

This example prints out the current Windows version number.

```
print "Currently using Windows "; winversion
```

## XONXOFF Statement

Used to change the state of the Xon/Xoff (software flow control) setting.

### Syntax

```
XONXOFF onoff
```

### Remarks

This function turns on or off Xon/Xoff (software flow control) from a script.

### See also

DEVICE

### Example

This example shows how the XONXOFF command might be used to turn on software flow control in a logon script.

```
waitfor "UserID"
send "123456"
xonxoff on
```

## XOR Operator

XOR performs a bitwise logical exclusive-or operation between its operands.

### Syntax

```
op1 XOR op2
```

### Remarks

op1 and op2 are logical expressions or numeric values of any integral type (byte, integer, or long integer).

This is the logical truth table for the XOR operator:

| a | b | a XOR b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### See also

AND, EQV, IMP, NOT, OR

### Example

This example shows how the XOR operator is used to do a bitwise exclusive-or operation.

```
print "5 xor 6 is 3: "; 5 xor 6
```

# Using Dialog Boxes

The *QmodemPro for Windows* script language allows you to create Windows dialog boxes that you can use to gather information from the user, display status information, and so on.

There are two steps that are necessary to create dialog boxes using the *QmodemPro for Windows* script language:

- Create the dialog box template.
- Create a dialog box variable based on the template.

## Creating a Dialog Box Template

A dialog box template declaration is very similar to a user defined type declaration. Here is the syntax for a dialog type declaration:

```
DIALOG dialogtype x, y, w, h
  [CAPTION caption]
  [FONT size, fontname]
  [integer-field AS CHECKBOX title, id, x, y, w, h]
  [integer-field AS COMBOBOX id, x, y, w, h]
  [CTEXT title, id, x, y, w, h]
  [DEFPUSHBUTTON title, id, x, y, w, h]
  [string-field as EDITTEXT id, x, y, w, h]
  [GROUPBOX title, id, x, y, w, h]
  [integer-field AS LISTBOX id, x, y, w, h]
  [LTEXT title, id, x, y, w, h]
  [PUSHBUTTON title, id, x, y, w, h]
  [integer-field AS RADIOBUTTON title, id, x, y, w, h]
  [RTEXT title, id, x, y, w, h]
  ...
END DIALOG
```

The DIALOG keyword is used to introduce a dialog box declaration. Following this keyword is the name of the dialog type. After the name are four numbers which define the position and size of the dialog:

| | |
|---|---|
| x | the horizontal position of the dialog relative to the top left corner of the screen |
| y | the vertical position of the dialog relative to the top left corner of the screen |
| w | the width of the dialog in "dialog units" |
| h | the height of the dialog in "dialog units" |

Notice that the width and height of the dialog are specified in "dialog units" — these are units that are based on the size of the font used in the dialog. A horizontal dialog unit is approximately 1/4 of the average width of the characters in the font, and a vertical dialog unit is 1/8 of the height of the characters in the font. Dialog units are specified in this way to avoid scaling problems when changing the font used for the dialog box.

Most dialog boxes will include a CAPTION statement that includes the title of the dialog box. If no CAPTION statement is included, the title will be blank. For example,

```
caption "My dialog box"
```

The FONT statement allows you to change the font used to display the dialog box. Any point size and face name may be specified here — the actual font used depends on the fonts installed in your Windows system (some types of fonts, like the Script font, available elsewhere in Windows are not available for use in dialog boxes). For example, to use 10 point Arial:

```
font 10, "Arial"
```

The remainder of the commands allowed in dialog box definitions define what are called "controls". Controls have some common features — each control must have a unique positive ID number or −1 if an ID number is not necessary, and must have a position and size defined by the x, y, w, h

parameters. Specific features and requirements of each of the control types are discussed in the following paragraphs.

## CHECKBOX

Defines a check box control that can be set either on or off. The title parameter defines the caption that appears to the right hand side of the check box itself. A positive ID number is required for this control type. The integer field associated with this control is 1 if the check box is checked, otherwise it is zero.

## COMBOBOX

Defines a combo box control that offers a list of choices. A positive ID number is required for this control type. The integer field associated with this control indicates the number of the choice that was picked. To fill the combo box with choices, a dialog INIT procedure must be defined (see below).

## CTEXT

Defines a centered-text control that displays text in the dialog box. The title defines the text to display. An ID number is not required for this control so it should be set to –1.

## DEFPUSHBUTTON

Defines the "default" push button in the dialog (the default push button is the one that gets pressed when the user presses Enter). The title defines the text that will appear on the face of the button. A positive ID number is required for this control type. There should only be one DEFPUSHBUTTON in the dialog box template.

## EDITTEXT

Defines a text edit control that allows the user to enter or edit text. A positive ID number is required for this control type. The string field associated with this control is the actual text that appears within the edit control.

## GROUPBOX

Defines a shaded box control that can be used to group controls together visually. An ID number is not required for this control so it should be set to -1.

## LISTBOX

Defines a list box control that offers a list of choices. A positive ID number is required for this control type. The integer field associated with this control indicates the number of the choice that was picked. To fill the list box with choices, a dialog INIT procedure must be defined (see below).

## LTEXT

Defines a left-justified text control that displays text in the dialog box. The title defines the text to display. An ID number is not required for this control so it should be set to -1.

## PUSHBUTTON

Defines a push button in the dialog. The title defines the text that will appear on the face of the button. A positive ID number is required for this control type. If the ID number is not IDOK or IDCANCEL, then a response function should be defined for the push button so that some action can be taken when the button is pressed (IDOK and IDCANCEL automatically cause the dialog box to close and return IDOK or IDCANCEL to the script).

## RADIOBUTTON

Defines a radio button control that can be set either on or off, with the additional property that only one radio button in a dialog may be "on" at any time. The title parameter defines the caption that appears to the right hand side of the radio button itself. A positive ID number is required for this control type. The integer field associated with this control is 1 if the radio button is checked, otherwise it is zero.

## RTEXT

Defines a right-justified text control that displays text in the dialog box. The title defines the text to display. An ID number is not required for this control so it should be set to -1.

After each of the desired controls is defined, the dialog box template is ended with the END DIALOG keywords.

## A Word About Control IDs

The positive ID numbers used by many of the above control types are used internally by Windows to manage the dialog box once it is displayed on the screen. For this reason, certain rules should be followed to ensure that the dialog box is displayed properly.

Some ID numbers with low values are reserved by Windows and should not be used except where they are intended to be used. To avoid problems, a good place to start numbering your controls such as check boxes and edit controls is at number 101 and continuing upwards. ID numbers should never be used more than once in the same dialog box.

Buttons generally follow the same rules as other controls except for the special ID values IDOK (equal to 1) and IDCANCEL (equal to 2). When the IDOK or IDCANCEL values are used with a button, that button will automatically cause the dialog box to close and return the corresponding ID number. Many dialog boxes just have two buttons, Ok and Cancel, and these ID values provide a quick way to make those buttons work as expected. Buttons defined with other ID values will not automatically cause the dialog box to close - to perform some action requires a dialog box response function (below).

## Using Dialog Box Variables

Once a dialog box template has been constructed and defined, it is easy to make a dialog box appear on the screen. Suppose that a dialog box template called MyDialog has been defined. The next step is to define a variable based on that template:

```
dim d as MyDialog
```

The variable d now refers to a dialog box variable based on the template. To make the dialog box appear on the screen, use the DIALOGBOX function:

```
dim result as integer
result = dialogbox(d)
```

The variable result will contain the ID of the button used to close the dialog (this can be used to determine whether the user pressed OK or Cancel, for example).

Normally a dialog box will appear with all fields blank and all check boxes and radio buttons turned off. If you would like to "preload" the dialog box with certain information, just assign the data to fields in the dialog box just as with a user defined type:

```
dim d as MyDialog
d.name = "John Doe"
call dialogbox(d)
```

In the above example the name field will be preloaded with the string "John Doe". After the dialog box has been executed, the name field will contain whatever the user placed in the corresponding edit field on the screen.

## Using Dialog Box Response Functions

For more flexibility in creating dialog boxes, *QmodemPro for Windows* allows "dialog box response functions" to be declared. These are functions in the script program that are executed when certain events occur while the user is using the dialog box.

This is the syntax for a dialog box response function declaration:

```
FUNCTION dialogtype.{fieldname | ID(id)} AS INTEGER
  ...
END FUNCTION
```

Dialog box response functions can respond to the following types of events:

- The changing of a check box or radio button
- The changing of a combo box or list box selection
- The changing of an edit text field
- The pressing of a button or default button

To define a response function for the first three types of events, use the following form of the declaration:

```
FUNCTION dialogtype.fieldname AS INTEGER
  ...
END FUNCTION
```

This function will be called whenever the associated field changes. Each of the fields in the corresponding dialog variable are available to this function directly without the need to refer to the actual dialog variable. For example, assume the following dialog box declaration:

```
dialog mydialog 50, 50, 150, 50
  caption "Test dialog"
  username as edittext 101, 20, 10, 110, 12
  guest as checkbox "Use GUEST account", 102, 20, 30, 110, 10
end dialog
```

Suppose we want to define a response function for the "guest" checkbox that automatically modifies the "username" field. Declare the response function like this:

```
function mydialog.guest as integer
  if guest then
    username = "Guest"
  else
    username = ""
  end if
end function
```

Now, whenever the "guest" button is turned from off to on, the "username" field will be automatically modified to contain the string "Guest". Whenever the "guest" button is turned from on to off, the "username" field will be cleared.

To define a response function for a button, you must use the following syntax for the response function declaration:

```
FUNCTION dialogtype.ID(id) AS INTEGER
  ...
END FUNCTION
```

This function will be called whenever the button associated with the given id is pressed.

Each of these dialog response functions returns an integer. This integer is used by the dialog manager in the following way: If the return value is nonzero, the dialog is closed and the DIALOGBOX function returns the value returned from the dialog box function. If the return value from a response function is zero, the dialog will not be closed.

Unlike a normal function, the return value from a dialog box response function is activated by assigning to the special variable DIALOGRESULT within the function. This is instead of assigning to the name of the function as in a regular user defined function.

## The Dialog Init Function

There is one more special dialog response function that is called during dialog initialization before the dialog box is displayed. This dialog response function is declared like this:

```
SUB dialogtype.INIT
  ...
END SUB
```

This function can be used to initialize any fields within the dialog, or to fill a list box or combo box with items. Note that this is the first place during

dialog box creation that you are allowed to fill a list box or combo box with items.

To fill a list box or combo box with items, use the AddListBoxItem or AddComboBoxItem subroutines like this:

```
call AddListBoxItem(HWindow, 102, "choice 1")
```

In this example, HWindow is a special variable available within a dialog box response function that refers to the Windows dialog box. 102 is the ID value for the list box that we are working with, and the "choice 1" string is the string to add to the list box. The AddComboBoxItem works similarly.

Since the HWindow identifier used in this function is really a hidden data member of the dialog structure, the AddListBoxItem and AddComboBoxItem subroutines can only be called from the dialog box INIT function or a dialog box response function.

# Using DLL Functions

Microsoft Windows provides special libraries of routines called dynamic-link libraries (DLLs) that let applications share code and resources. *QmodemPro for Windows* script language allows you to call functions that reside in external DLLs.

In order to call a function within a DLL from a script, the following rules must be observed:

The function must use the Pascal calling convention. This means that the name of the function is case insensitive and is not preceded by an underscore character. Also, the parameters to the function are pushed from left to right and are removed from the stack by the function, not the caller.

The function cannot have parameters passed by value other than INTEGER, LONG, STRING, or a pointer to a user defined type.

If the function returns a value, it must be of type INTEGER, LONG, or STRING.

## Declaring DLL Functions

DLL functions must be declared using the following syntax of the DECLARE statement:

```
DECLARE {SUB | FUNCTION} name LIB "libname" [ALIAS "aliasname"]
[(arglist)] [AS type]
```

The keyword LIB indicates the name of the DLL in which the function resides. This keyword must be followed by a literal string that names the DLL (the .DLL extension need not be supplied).

Normally, the function is expected to be in the DLL under the same name as the function is declared in the DECLARE statement. In some cases this may be undesirable, as in the case where the name of the function in the DLL duplicates a reserved word or an identifier already declared in your script. In this case the ALIAS clause allows you to use a different name for

the function in the script language DECLARE statement, but the alias name is the actual name of the function in the DLL.

## Calling DLL Functions

Once a DLL function is declared, it can be called just like any other subroutine or function. For example, to use the MessageBox function, declare it as follows:

```
declare function MessageBox lib "user" (hwnd as integer, message as
string, caption as string, flags as integer) as integer
```

Once declared as above, you can use the MessageBox function like this:

```
dim i as result
result = MessageBox(0, "Save changes?", "Question", MB_YESNOCANCEL)
```

The above example works provided that MB_YESNOCANCEL is defined to be the same as its value in the WINDOWS.H file supplied with the Windows SDK.

In some cases, you may not care about the return value of a DLL function. Instead of using a dummy variable to accept the return value, you can eliminate the return value entirely by declaring the DLL function as a SUB instead of a FUNCTION.

## Parameters Passed to DLL Functions

The following specific rules are used when passing parameters to DLL functions:

- INTEGER and LONG — passed by value instead of reference.
- STRING — a pointer to a null terminated string is passed.
- user defined type — a pointer to the user defined type is passed.

When returning an INTEGER or a LONG from a DLL function, the standard parameter passing rules are used. When returning a STRING, the function is expected to return a far pointer to a null-terminated string. *QmodemPro for Windows* will automatically make a copy of the string for the return value.

# 3 - Script Examples

*Few things are harder to put up with than the annoyance of a good example.*

**Mark Twain**

# In this chapter

Script Examples ..... 257
Simple Logon ..... 257
Mail Pickup ..... 258
Other examples ..... 262

# Script Examples

The previous sections discussed the *QmodemPro for Windows* script language by explaining commands and their individual operation. This section takes a different perspective and reviews more extensive complete script examples. By discussing several scripts and their components it may be easier to follow the flow and operation of each command.

The following examples illustrate how the script language can be used in a number of applications.

## Simple Logon

The first example is a simple logon script to a BBS system, generated by QuickLearn:

```
SEND "ATDT 873-2400"
WAITFOR "s your first name? "
SEND "Rick Heming"
WAITFOR "              ][15D"
SEND "mypassword"
```

Line 1 was generated when we began typing in the terminal window. We typed the standard ATDT command to dial a phone number and QuickLearn dutifully recorded it all, placing quotes around the exact text we typed.

Line 2 was generated after the BBS answered. It asked for the callers first name, and when we began typing the name it looked back on what it had been recording and found the last 20 characters were "s your first name? ".

Line 3 was also generated when we began typing the name, telling *QmodemPro for Windows* what name to send.

Line 4 was generated exactly like line 2, and contains the last 20 characters of the BBS line that asks for the caller's password. In this case,

the last 20 characters are mostly spaces since the prompt (with ANSI color codes included) is:

```
Password ? [                ][15D
```

The brackets are separated by 15 spaces and the last 5 characters represent the ANSI commands to move the cursor back to the position immediately following the first bracket. Regardless of the position of the cursor on the screen, the characters were received as indicated above, and the last 15 characters are those in the WAITFOR command in line 4.

Line 5 was generated when the password was typed and QuickLearn was then stopped.

The reply to the password prompt the text in quotes could be replaced by the system function PASSWORD which would allow the same script to be used with different Phonebook entries, provided they asked for the same information in the same order.

## Mail Pickup

The following script is much more complex and makes use of additional features. It is used to call the MSI HQ BBS and download messages using the off-line mail reading door. The comments explain each line, and a discussion will follow.

```
'
' This script calls MSI HQ BBS and uploads and downloads mail
' using the Tomcat mail door online. It is designed to be run
' as a script linked to a phonebook entry since it does not
' dial the number.
when match "-Pause-" do send
    ' set up a when on "-Pause-" to take care of the case where
    ' the sysop has a prelog screen with a pause in it.
waitfor "first name"
    ' wait for name and password prompts and send the correct
    ' responses.
```

```
send LASTCONNECTUSERID
    ' after getting the prompt, send the user name by calling
    ' the system function LASTCONNECTUSERID which uses the entry
    ' as defined in the phonebook entry under User ID.
waitfor "password"
delay 0.2
    ' wait for the password prompt. This time we ignore the ANSI
    ' codes that come immediately after the text 'password'
    ' since we use a delay (.2 seconds) before sending our
    ' password text.
send LASTCONNECTPASSWORD
    ' then send the password using the system function that gets
    ' the information from the phonebook entry.
when match "] to continue" do send
when match "view the bulletin menu" do send "N" ;
when match "have new personal mail" do send "C" ;
when match "protocol menu [" do send "Z" ;
    ' set up a number of match type whens to provide answers to
    ' various prompts. A blank 'send' sends a C/R only. Also, if
    ' the user record does not have hot-keys enabled the
    ' semicolons at the end of each line must be removed to
    ' include a C/R.
when quiet 10 do send
    ' a when quiet is used to press Enter whenever we don't get
    ' any characters for a particular amount of time, 10 seconds
    ' in this case.
timeout 60
    ' set the default termination timeout to 60 seconds - if we
    ' don't get what we're looking for in that time then check
    ' below in the "catch err_timeout" section.
```

```
    ' next we make our way to the Tomcat main menu, look for the
    ' word 'command' to tell us we are at the Main Menu, then
    ' send a 'M' to go to the Message Menu. Then send 'T' to
    ' enter the Tomcat mail program.
waitfor "command"
send "M"
waitfor "command"
send "T"
waitfor "tomcat menu ["
    ' now we're at the Tomcat main menu.
if exists("c:\qmwin\upload\mustang.rep") then
    ' check to see if we have a REP file to upload in our path.
    ' If so, go ahead and send it.
  send "U"       ' upload start command
  waitfor "now"
    ' it is ready for you to begin the upload but assumes you
    ' have Zmodem set as a default transfer protocol.
  if upload("c:\dl\mustang.rep", Zmodem) = 0 then
    ' the command above starts the actual transfer with the
    ' upload (filename, protocol) statement, and also checks the
    ' success of the upload by comparing the result of the
    ' statement (0 indicates success, non-zero indicates
    ' failure).
     del "c:\dl\mustang.rep"            ' if success, delete it
  end if      ' if not a success, it is kept.
  waitfor "tomcat menu ["
end if    ' if mustang.rep doesn't exist it comes here.
    ' now download the new mail.
send "D"
waitfor "would you like", 200
```

```
    ' we provide a timeout of 200 seconds on the 'waitfor'
    ' that waits for the prompt at the end of the scan, that
    ' way if this takes a long time we won't just abort.
send "Y"
    ' then respond to the download/quit/logoff prompt.
waitfor "now", 120
    ' wait until the file is ready for download.
del "c:\dl\mustang.qwk"
    ' delete the current .QWK packet if there is one.
call download("c:\dl", Zmodem)
    ' call the download statement with a path and protocol.
waitfor "tomcat menu ["
    ' wait for Tomcat to return to its menu.
send "G"
    ' and send the command to log off.
end
    ' at this point the script will end and return to
    ' terminal mode.
' The following is a catch clause for the main program body, it
' handles timeout errors that are not otherwise handled.
' If we get a timeout here, then there's not much we can do so
' we just hang up and abort.
catch err_timeout
  hangup
  end
    ' end of timeout catch block
```

The script above automates the process of mail download from a BBS and handles several possible situations that could develop during the connection. It could easily be modified to handle additional situations, such as retrying the upload or download routines 3 times if they fail.

## Other examples

There are a number of additional scripts included with your *QmodemPro for Windows* package. Each is fully commented and provides additional examples of how the script language may be used.

One of the best ways to learn scripting is to load the examples into the debugger and follow the program flow as it takes place.

# 4 - Debugging

*A man of genius makes no mistakes. His errors are volitional and are the portals of discovery.*

**James Joyce**

# In this chapter

Debugging A Script ..... 265
Compiler Errors ..... 268
Runtime Error Messages ..... 280

# Debugging A Script

As we indicated earlier, regardless of the simplicity of a script or the capabilities of the programmer, scripts are seldom written perfectly on the first attempt.

Compiler errors are caused by improper statements, syntax, or program constructs, and can usually be located and corrected fairly easily. In contract, script code that compiles correctly but causes execution or run-time errors can often be difficult to locate. Even harder are logic errors that both compile and run properly, but produce incorrect results or flawed program flow. In these cases, a script debugger can be of invaluable assistance.

A compiled script language has advantages over an interpreted script language but the fact that its executable code is not connected directly to its source can make it difficult to relate a run-time error to the proper source statement.

## Running a debug session

Starting the SLIQ debugger is done by selecting the **Script/Debug** menu choice, which displays the standard file selection dialog box. Once a script file is selected, it is recompiled regardless whether an executable .SCC file exists. During this compile session an additional debug info file is created in the scripts directory. The debug info file uses the extension .SCD and contains references to other files called by the script, line number offsets into the executable file, and type and location information on variables. The debug info file is used only by the debugger and may be erased after debugging is completed.

Once the compile session is complete the debug screen is displayed. It consists of two main sections, the source window and the watch window.

Both the source and watch windows can be arranged and resized within the debugger window, but they cannot be placed outside of the window. If the watch window is not being used it may be minimized, but it cannot be closed.

## The Source Window

The source window displays the original source script file. In the left hand margin of the window there is an area where two special symbols are displayed:

An arrow in the left hand column indicates that the current execution position of the script is stopped on the indicated line.

A red circle with a B inside it indicates that the associated line has a breakpoint set on it. Breakpoints are discussed below.

## The Watch Window

The watch window is used to watch the values of variables while your script program is executing. If you watch a variable in this window using the Watch button on the toolbar, its value will be updated in the window whenever it is modified by the script program.

## The Step Button

The Step button causes the debugger to execute one script statement. If the script statement is a subroutine or function call, the debugger will stop on the first statement of the subroutine or function body.

## The Jump Button

The Jump button is similar to the Step button, except that if the current statement calls a subroutine or function, the statements within the subroutine or function will automatically be skipped.

## The Return Button

The Return button causes the debugger to execute statements continuously until the return from the current subroutine or function. This button is useful if you Stepped into a subroutine or function and meant to Jump over it.

## The Go Button

The Go button continuously executes the script until the script ends or a breakpoint is encountered.

## The Stop Button

The Stop button can be pressed at any time during script execution to cause the debugger to stop the script program at the next possible point. Note that the script may not stop immediately, but it will stop when the currently executing script statement has finished.

## The Watch Button

The Watch button is used to select a variable to watch in the Watch window. If the cursor in the Source window is positioned over a variable name when the Watch button is pressed, that variable will automatically be watched. If not, a selection box will be presented that will allow you to either pick a variable to watch or type in a variable name.

## Setting Breakpoints

To set a breakpoint in the Source window, click on the desired line using the right mouse button. To clear the breakpoint, click on the line with the right mouse button again. Note that the breakpoint indicator may not appear on the same line that you click on — it will appear on the next line of the source that contains an executable statement.

# Compiler Errors

When the compiler detects an error compiling a script, it will automatically open up the *QmodemPro for Windows* editor and place the cursor at the position the error occurred. An error message will appear in a dialog box. The error messages that can occur are listed below in numerical order.

## Error 1: Error opening input file

This error occurs when the script compiler cannot open an input file, whether it is the original file or an included file.

## Error 2: Error creating output file

This error occurs when the script compiler cannot create the script output file for some reason.

## Error 3: Syntax error

This error indicates a generic problem with the script program. Check with the documentation for the statement you are trying to use to make sure the statement is formed correctly.

## Error 4: End of statement expected

This error can occur if you have placed extra information on a line that is not part of the statement on that line.

## Error 5: Number too large

Integer numbers must be in the range -2147483648 to 2147483647.

## Error 6: Error in floating point number

The script compiler has encountered a number which it determines should be a floating point number, but it is not in a correct format.

## Error 7: String not terminated

A double quote character appears on the line without a closing double quote character.

## Error 8: = expected

An equals sign is expected here. You may get this error if you misspell a script statement — the script compiler will not recognize the word and may assume that it is a variable you are trying to assign a value to.

## Error 9: ( expected

An open parenthesis is expected here. You may be trying to call a function with no arguments when it actually requires arguments.

## Error 10: ) expected

A close parenthesis is expected here. One of the common places where this error occurs is if you omit the CALL keyword when calling a subroutine, but try to place parentheses around the parameter list anyway. This error will occur at the end of the first argument.

## Error 11: , expected

A comma is expected here. You may be trying to call a function with fewer parameters than it requires.

## Error 12: Type mismatch

A type mismatch error indicates that you are using a variable or expression that is the wrong type for the expression expected there.

## Error 13: THEN expected

The THEN keyword is expected after the condition in an IF statement.

## Error 14: ELSE not allowed here

The ELSE keyword is only permitted after an IF or ELSEIF clause. If an ELSE clause for the current IF statement has already been encountered, another ELSE clause is not permitted.

## Error 15: ELSEIF not allowed here

The ELSEIF keyword is not permitted without a preceding IF statement, and it is also not permitted after an ELSE clause.

## Error 16: UNTIL or WHILE expected

The UNTIL or WHILE keyword is expected after a DO or LOOP keyword.

## Error 17: LOOP not allowed here

The LOOP keyword is only permitted after a matching DO statement.

## Error 18: EXIT DO not within DO ... LOOP

The EXIT DO statement must occur within a DO ... LOOP construct.

## Error 19: EXIT FOR not within FOR ... NEXT

The EXIT FOR statement must occur within a FOR ... NEXT loop.

## Error 20: DO, FOR, SUB, or FUNCTION expected

The EXIT keyword must be used with one of the listed keywords.

## Error 21: Duplicate label declaration

The label you are trying to define has already been defined within the current function.

## Error 22: LOOP expected

The LOOP keyword is expected at the end of a DO ... LOOP.

### Error 23: END IF expected

The END IF keywords must follow an IF statement.

### Error 24: WEND expected

Each WHILE statement must match up with a corresponding WEND statement.

### Error 25: Variable expected

A variable is expected wherever an assignment occurs.

### Error 26: WEND not allowed here

Each WEND statement must have a corresponding WHILE statement.

### Error 27: Function expected

This error means that you tried to call a subroutine as a function. Since subroutines cannot return a value, this is not possible.

### Error 28: Subroutine name expected

A subroutine name is expected after the CALL keyword.

### Error 29: FUNCTION or SUB expected

Either the FUNCTION or SUB keyword is expected after the DECLARE keyword.

### Error 30: Identifier expected

An identifier is expected here. This generally means that you are trying to declare an identifier but none was found.

### Error 31: Type character illegal for SUB

Subroutine declarations, since they do not return a value, cannot have a type specification.

## Error 32: AS expected

The AS keyword is expected here. Since you need an AS keyword, the preceding identifier needs a type declaration.

## Error 33: Type name expected

A type name is expected after the AS keyword.

## Error 34: Type specification illegal

A type specification is not allowed in the indicated case.

## Error 35: Parameter list type mismatch

A function declaration and the corresponding definition must match in the number and type of arguments.

## Error 36: Parameter list has fewer items than declaration

A function declaration and the corresponding definition must match in the number and type of arguments.

## Error 37: Parameter list has more items than declaration

A function declaration and the corresponding definition must match in the number and type of arguments.

## Error 38: Nested FUNCTIONs or SUBs not allowed

User defined subroutines and functions must not be declared without properly ending the previous subroutine or function.

## Error 39: EXIT FUNCTION/SUB not allowed in main code

EXIT FUNCTION and EXIT SUB can only appear within a subroutine or function, not the main program body.

## Error 40: AS not allowed here

This error is caused by using the AS keyword in a place where it is not permitted.

### Error 41: Duplicate definition

The subroutine or function you are trying to define has already been defined once before.

### Error 42: Duplicate identifier

The identifier you are trying to declare has already been used as a variable, constant, subroutine or function name, or type name.

### Error 43: Constant expression required

A constant expression is required here. For example, when declaring an array, the array dimension must be a constant.

### Error 44: Array variable expected

This error is most often caused by trying to use a subscript with a variable that has not been declared as an array.

### Error 45: Integer expression required

An integer expression is required here. For example, when dimensioning an array, an integer value is required for the dimension value.

### Error 46: Invalid array bounds

The lower bound of an array must be less than or equal to the upper bound.

### Error 47: Reserved for future use

### Error 48: File mode expected

One of the keywords INPUT, OUTPUT, APPEND, BINARY, or RANDOM must follow the FOR keyword in an OPEN statement.

### Error 49: Reserved for future use

### Error 50: Reserved for future use

### Error 51: LEN expected

The LEN keyword is expected at the end of an OPEN statement.

### Error 52: TO expected

The TO keyword is expected after the starting value in a FOR statement.

### Error 53: NEXT expected

The NEXT keyword is expected after a FOR loop.

### Error 54: NEXT not allowed here

The NEXT keyword must be preceded by a matching FOR statement.

### Error 55: NEXT without FOR

You have used a NEXT keyword with a variable that does not match the variable used in the corresponding FOR keyword.

### Error 56: CASE expected

The keyword CASE is required after the keyword SELECT.

### Error 57: END SELECT expected

The END SELECT keywords must follow a SELECT CASE block.

### Error 58: CASE not allowed after CASE ELSE

The CASE ELSE clause in a SELECT CASE statement must be the last case clause in the statement.

### Error 59: ELSE or IS expected

If a CASE keyword in a SELECT CASE statement is not followed by an expression, it must be followed by an ELSE or IS keyword.

## Error 60: Statements not allowed here

Statements are not allowed between a SELECT CASE statement and the first CASE clause. Also, statements are not permitted within a user defined type declaration or a dialog box template definition.

## Error 61: =, <>, <, <=, >, or >= expected

In a CASE clause within a SELECT CASE statement, the keyword IS must be followed by one of the relational operators listed.

## Error 62: END SUB or END FUNCTION expected

The END SUB or END FUNCTION keywords are expected here.

## Error 63: EXIT FUNCTION expected

If the EXIT keyword appears within a function, it should be an EXIT FUNCTION statement.

## Error 64: EXIT SUB expected

If the EXIT keyword appears within a subroutine, it should be an EXIT SUB statement.

## Error 65: END FUNCTION expected

The END FUNCTION keywords must follow a function declaration.

## Error 66: END SUB expected

The END SUB keywords must follow a subroutine declaration.

## Error 67: Undeclared identifier

Identifiers must be declared before using them. This error is most often caused by using a variable before having a DIM statement that declares it.

## Error 68: END TYPE expected

The END TYPE keywords must follow a user defined type declaration.

## Error 69: Duplicate type definition

The type name you are trying to declare has already been declared.

## Error 70: Type character not allowed

A type specification character is not allowed here.

## Error 71: User defined type variable expected

You are trying to use the dot notation to access a field of a user defined type, but the variable to the left of the dot is not a user defined type variable.

## Error 72: Field name expected

A field name must appear after a period when accessing a particular field of a user defined type.

## Error 73: Dynamic strings not allowed in user defined types

Variable sized strings are not permitted in a user defined type declaration. Use a fixed length string instead.

## Error 74: Parameter required

This error occurs when you call a function that requires more parameters than you have supplied.

## Error 75: Default arguments not allowed in function definition (use DECLARE)

Default arguments to function can only appear within DECLARE statements, not the actual function body definition.

## Error 76: WHEN condition expected

The WHEN condition must be one of MATCH, QUIET, or TIME.

## Error 77: DO expected

The keyword DO is required after the condition in a WHEN statement.

## Error 78: END WHEN expected

The END WHEN keywords must follow a WHEN statement.

## Error 79: END DIALOG expected

The END DIALOG keywords must follow a dialog type declaration.

## Error 80: Control type expected

You cannot declare fields of a normal type within a dialog box definition; all fields must be dialog box control types.

## Error 81: Dialog variable expected

The DIALOGBOX function requires a variable declared based on a dialog box template as an argument.

## Error 82: Field variable required with this control type

An associated field variable is required with the CHECKBOX, COMBOBOX, EDITTEXT, LISTBOX, or RADIOBUTTON control types.

## Error 83: Field variable not allowed with this control type

An associated field variable is not permitted with the CTEXT, DEFPUSHBUTTON, GROUPBOX, LTEXT, PUSHBUTTON, or RTEXT control types.

## Error 84: Dialog type expected

When declaring a dialog box response function, the name of the type used in the function name must be the name of a dialog box type.

## Error 85: Control ID number expected

A control ID number is required when declaring a response function for an unnamed control.

## Error 86: Dialog event must be a function returning integer

Dialog response functions must return an integer value to the dialog box manager so that it can tell whether or not to close the dialog box.

## Error 87: Dialog init must be a subroutine

The dialog initialization function must be a subroutine — it cannot return a value to the dialog box manager.

## Error 88: String required after ALIAS

A literal string is required after the ALIAS keyword in a DLL function declaration.

## Error 89: String required after LIB

A literal string is required after the LIB keyword in a DLL function declaration.

## Error 90: LIB only allowed with DECLARE

The LIB keyword is only allowed within a function declaration using DECLARE. You have probably omitted the DECLARE keyword.

## Error 91: Function result must be numeric or string

Function results can only be a numeric or string type, not a user defined type.

## Error 92: Fixed length string parameter not allowed

Fixed length string parameters to subroutines and functions are not permitted. Use a variable length string instead.

## Error 93: Function already defined

The function you are trying to define has already been defined.

## Error 94: CATCH not allowed here

A CATCH statement is only permitted at the end of a subroutine or function (or at the end of the main program body). A CATCH statement is not permitted after a CATCH ALL statement.

## Error 95: Label "label" not declared in block

You have used the GOTO or GOSUB statement with a label that was not declared within the same statement block as the GOTO or GOSUB statement.

## Error 96: Function "function" not declared in script

You have declared a function but have not provided a definition for the function.

## Error 97: INPUT or OUTPUT expected

The keyword INPUT or OUTPUT or both is required after the FLUSH command.

## Error 98: Positive control ID required for this control type

Control types such as combo boxes and edit text fields require positive control IDs. See the section on Using Dialog Boxes in chapter 2 for more information.

## Error 99: ENTRY, GROUP, MANUAL, or SEARCH expected

One of the above keywords is required after the DIAL command.

## Error 100: No local variables can be used in a WHEN block

Because WHEN statements are active until explicitly cleared, you cannot use any local variables of an enclosing procedure because the procedure may exit while your WHEN statement is still active. You may access global variables (variables declared outside any subroutine or function) from within an WHEN block.

# Runtime Error Messages

There are certain errors that cannot be detected at compile time and can occur only while running a script. These errors will pop up a box on the screen with the error message and terminate running the script. Some of these errors can be caught with the CATCH clause — these errors are noted below.

## "Invalid opcode"
## "Unexpected end of file"

Catch name: (cannot be caught)

These errors both mean that there was a problem with the script's .SCC file and the script probably needs to be recompiled.

## "STOP encountered"

Catch name: (cannot be caught)

This error means that a STOP statement was encountered while running the script.

## "RETURN without GOSUB"

Catch name: (cannot be caught)

This error means that a RETURN statement was encountered without a corresponding GOSUB statement. Be sure that the GOSUB and RETURN statements in your script are properly balanced.

## "Math error"

Catch name: ERR_MATH

This error indicates that an arithmetic error has occurred (such as division by zero, taking the logarithm of a nonpositive number, and taking the square root of a negative number).

### "Array subscript out of bounds"

Catch name: ERR_ARRAYSUBSCRIPT

An attempt was made to access an array element outside the bounds of the array.

### "Library not found"

Catch name: ERR_LIBRARYNOTFOUND

An attempt was made to call a function in an external DLL that could not be loaded.

### "Function not found"

Catch name: ERR_FUNCTIONNOTFOUND

An attempt was made to call a function in an external DLL and the named function could not be found.

### "Invalid file number"

Catch name: ERR_INVALIDFILENUMBER

An attempt was made to access an invalid or unused file number in a file access statment such as PRINT or GET.

### "Error opening file"

Catch name: ERR_FILEOPEN

The file named in an OPEN statement could not be opened (most likely, the file could not be found on disk).

### "Path not found"

Catch name: ERR_PATH

This error occurs when a CHDIR, CHDRIVE, MKDIR, or RMDIR statement accesses a drive or directory that does not exist.

## "Error renaming file"

Catch name: ERR_FILERENAME

An error occurred during a file rename operation - either the original file does not exist or the destination name is already in use.

## "Communications timeout"

Catch name: ERR_TIMEOUT

The time limit has expired on a WAITFOR statement.

## "WHEN ... GOTO executed from noncurrent scope"

Catch name: (cannot be caught)

If a WHEN statement contains a GOTO statement, then the destination of the GOTO statement can only be within the current scope (ie. within the same subroutine or function). Since a WHEN can be triggered at any time, this error can occur if you set up a WHEN statement with a GOTO within it, and then call another subroutine, but the WHEN statement is triggered while in the subroutine.

To avoid this situation, you can set up a global flag that the WHEN statement will set, and then test for this flag during your script. This prevents directly executing a GOTO from within a WHEN statement.

# Index


$

$INCLUDE Directive, 131

(

( expected, 269

)

) expected, 269

+

+ (Concatenation), 34

,

, expected, 269

=

= expected, 269
=, <>, <, <=, >, or >= expected, 275

8

8th Bit Strip, 215

A

ABS Function, 35
ACTIVATE Statement, 36
ADDENTRY Statement, 37
ADDLFTOCR Function, 39
Alias, 254
AND Operator, 40
ANSI, 102, 258, 259
Answer, 44
Answer off, 44
Answer on, 44
Array subscript out of bounds, 281
Array variable expected, 273
Arrays, 20
AS expected, 272
AS not allowed here, 272
ASC Function, 42
ASCII, 11, 13, 18, 42, 61, 96, 177, 188, 196, 214, 227
ATN Function, 43
AUTOANSWER Statement, 44
Automated logon, 10
automated sessions, 10

B

Batch file operation, 201
Baud rate, 198
BBS, 1, 10, 38, 96, 177, 196, 227, 257, 258, 261
BEEP Statement, 45
Birthdate, 21
BMP, 195, 232
Break, 46
Break length, 46
BREAK Statement, 46
Breakpoints, 266, 267
BYTE Type, 47
BYVAL Identifier, 48

C

CALL Statement, 50
Calling DLL Functions, 254
Cancel, 89, 248, 249
CAPTURE Statement, 52
CARRIER Function, 53
CASE expected, 274
CASE not allowed after CASE ELSE, 274
CASE Statement, 54, 192
CATCH not allowed here, 279
CATCH Statement, 55
CHAIN Statement, 58
CHDIR Statement, 59

CHDRIVE Statement, 60
CHR Function, 61
Clear, 132
CLOSE Statement, 62
CLS Statement, 63
COLOR Statement, 64
columns, 146, 169
Comments, 18, 179
Communications timeout, 282
Compiler Errors, 268
Compiling, 9, 13
CONFIGCAPTUREFILE Function, 66
CONFIGDOWNLOADPATH Function, 67
CONFIGLOGFILE Function, 68
CONFIGSCRIPTPATH Function, 69
CONFIGSCROLLBACKFILE Function, 70
CONFIGTRAPFILE Function, 71
CONFIGUPLOADPATH Function, 72
Connection, 46
CONST Statement, 73
Constant expression required, 273
Constants, 18, 73
Control ID number expected, 277
Control type expected, 277
COPYFILE Function, 74
Copyright, 2
COS Function, 75
CSRLIN Function, 76
CURDIR Function, 77
CURDRIVE Function, 78

D

Data Terminal Ready, 199
DATE Function, 79
Debugging, 14, 263, 265
DECLARE Statement, 80
Declaring DLL Functions, 253
Default arguments not allowed in function definition, 276
DEL Statement, 81
DELAY Statement, 82
Design elements, 10
Device, 46, 125
DEVICE Statement, 83
DIAL ENTRY, 84
DIAL GROUP, 84, 86
DIAL MANUAL, 84
DIAL SEARCH, 84
DIAL Statement, 84
DIALNEXT Function, 86
Dialog Box, 21, 89, 244, 248, 249, 279
Dialog Box Response Functions, 249
Dialog Box Template, 244
Dialog Box Types, 21
Dialog Box Variables, 248
Dialog event must be a function returning integer, 278
Dialog Init Function, 251
Dialog init must be a subroutine, 278
DIALOG Statement, 87
Dialog type expected, 277
Dialog variable expected, 277
DIALOGBOX Function, 89
DIM Statement, 90
DO ... LOOP Statement, 92
DO expected, 277
DO, FOR, SUB, or FUNCTION expected, 270
DOORWAY Function, 94
Doorway mode, 94
DOSVERSION Function, 95
download, 10, 67, 96, 97, 177, 258, 260, 261
DOWNLOAD Function, 96
DTR, 125, 199
DUPLEX Function, 98
Duplicate, 270
Duplicate definition, 273
Duplicate identifier, 273
Duplicate label declaration, 270
Duplicate type definition, 276
Dynamic menus, 55, 56, 253
Dynamic strings not allowed in user defined types, 276

E

Echo conferences, 98, 127

Edit, 11, 12, 13
Edit a File, 13
EDITFILE Statement, 99
ELSE not allowed here, 270
ELSE or IS expected, 274
ELSE Statement, 100
ELSEIF not allowed here, 270
ELSEIF Statement, 101
Emulation, 181
EMULATION Statement, 102
Emulations, 76
END DIALOG expected, 277
END FUNCTION expected, 275
END IF expected, 271
End of statement expected, 268
END SELECT expected, 274
END Statement, 103
END SUB expected, 275
END SUB or END FUNCTION expected, 275
END TYPE expected, 275
END WHEN expected, 277
ENTRY, GROUP, MANUAL, or SEARCH expected, 279
ENVIRON Function, 104
Environment variable, 11, 104
EOF Function, 105
EQV Operator, 106
ERR_ARRAYSUBSCRIPT, 55, 281
ERR_FILEOPEN, 52, 55, 107, 281
ERR_FILERENAME, 55, 161, 282
ERR_FUNCTIONNOTFOUND, 55, 281
ERR_INVALIDFILENUMBER, 56, 281
ERR_LIBRARYNOTFOUND, 56, 281
ERR_MATH, 56, 148, 157, 208, 280
ERR_PATH, 56, 156, 185
ERR_TIMEOUT, 56, 233, 282
Error creating output file, 268
Error in floating point number, 268
Error messages, 280
Error opening file, 281
Error opening input file, 268
Error renaming file, 282
ERROR Statement, 107
EXISTS Function, 108
EXIT DO not within DO ... LOOP, 270
EXIT FOR not within FOR ... NEXT, 270
EXIT FUNCTION expected, 275
EXIT FUNCTION/SUB not allowed in main code, 272
EXIT Statement, 109
EXIT SUB expected, 275
EXP Function, 110
Expressions, 22

## F

Fax, 1
Field name expected, 276
Field variable not allowed with this control type, 277
Field variable required with this control type, 277
File, 13, 15, 52, 66, 68, 70, 71, 72, 144, 149, 171, 195, 273
File Definitions, 52, 149
File Manager, 15
File mode expected, 273
File transfers, 10
File/Open, 144
FINDFIRST Function, 111
FINDNEXT Function, 113
First call, 29
FIX Function, 114
Fixed length string parameter not allowed, 278
FLUSH Statement, 115
font, 17, 87, 195, 244, 245
FOR ... NEXT Statement, 116
For statement, 26
FREEFILE Function, 117
From, 195
Function, 279
Function already defined, 278
Function expected, 271
Function not found, 281
FUNCTION or SUB expected, 271
Function result must be numeric or string, 278
FUNCTION Statement, 118
Functions, 22, 28, 29, 80, 118, 249, 253, 254

G

GET Statement, 120
GIF, 155, 232
Go Button, 267
GOSUB ... RETURN Statement, 122
GOTO Statement, 124

H

Handles, 261
HANGUP Statement, 125
HEX Function, 126
high bit, 215
HOSTECHO Function, 127

I

Identifier, 271
Identifier expected, 271
Identifiers, 18, 275
IF Statement, 128
IMP Operator, 130
INCLUDE Directive, 131
Init, 251
INKEY Function, 132
INPUT or OUTPUT expected, 279
INPUT Statement, 133
INSTR Function, 134
INT Function, 135
Integer expression required, 273
INTEGER Type, 136
Invalid array bounds, 273
Invalid file number, 281
Invalid opcode, 280

J

Jump Button, 266

K

Keywords, 17
KILL Statement, 137

L

Label, 279
LASTCONNECTPASSWORD Function, 138
LASTCONNECTUSERID Function, 139
LCASE Function, 140
LEFT Function, 141
LEN expected, 274
LEN Function, 142
LET Statement, 143
LIB only allowed with DECLARE, 278
Library not found, 281
LOADPHN Statement, 144
LOC Function, 145
LOCATE Statement, 146
LOF Function, 147
log file, 68, 149, 209
LOG Function, 148
Log Toggle, 149
LOGFILE Statement, 149
Logon, 257
LONG Type, 150
LOOP expected, 270
LOOP not allowed here, 270
Loop statement, 24, 92, 151
LTRIM Function, 152

M

Mail Pickup, 258
Math error, 280
MAXIMIZE Statement, 153
MID Function, 154
MINIMIZE Statement, 155
MKDIR Statement, 156
MOD Operator, 157
Modem, 125, 182
Modem init, 182
MOUSECLICK Statement, 158
MOVE Statement, 159
MSGBOX Statement, 160
MSI HQ BBS, 38, 258
Multiple line if statement, 23

N

NAME Statement, 161

Nested FUNCTIONs or SUBs not allowed, 272
NEXT expected, 274
NEXT not allowed here, 274
NEXT Statement, 116, 162
NEXT without FOR, 274
No local variables can be used in a WHEN block, 279
NOT Operator, 163
Note, 13, 18, 21, 92, 99, 120, 172, 191, 218, 231, 232, 251, 267
Number too large, 268
Numbers, 17

## O

OCT Function, 164
OK, 18, 89, 160, 223, 249
Open, 144
OPEN Statement, 165
Options, 11, 13, 46, 52, 66, 67, 68, 69, 70, 71, 72, 76, 96, 125, 149, 171, 177
OR Operator, 166

## P

Parameter list has fewer items than declaration, 272
Parameter list has more items than declaration, 272
Parameter list type mismatch, 272
Parameter required, 276
Parameters Passed to DLL Functions, 254
parity, 198
password, 10, 37, 138, 257, 258, 259
Path Definitions, 11, 13, 67, 69, 96, 177
Path not found, 281
pause, 167, 258
PAUSE Statement, 167
PCX, 195
Phonebook, 12, 15, 258
PLAY Statement, 168
POS Function, 169
Positive control ID required for this control type, 279
Print, 171
PRINT Statement, 170
PRINTER Statement, 171
PUT Statement, 172

## Q

QuickLearn, 9, 10, 12, 13, 257, 258

## R

REAL Type, 174
RECEIVE Statement, 175
RECEIVEFAX Statement, 176
RECEIVEFILE Function, 177
REM Statement, 179
Reports, 169
Reset Emulation, 181
RESET Statement, 180
RESETEMULATION Statement, 181
RESETMODEM Statement, 182
Return Button, 266
RETURN Statement, 122, 183
RETURN without GOSUB, 280
RIGHT Function, 184
RIPscrip, 102, 158
RMDIR Statement, 185
RND Function, 186
RTRIM Function, 187
Run, 15
Running a script, 14
Runtime Error, 280

## S

Save, 254
SCREEN Function, 188
Script, 1, 5, 11, 13, 14, 15, 16, 31, 33, 245, 255, 265
Scripts, 9, 11, 12, 13, 14, 15, 16, 58
SCROLLBACK Statement, 189
SCROLLBACKRECORD Function, 190
SEEK Statement, 191
Select, 26
Select case statement, 26, 192
SEND Statement, 194
SENDFAX Statement, 195
SENDFILE Function, 196
SETCOMM Statement, 198

SETDTR Statement, 199
Setting Breakpoints, 267
SGN Function, 200
SHELL Statement, 201
Simple types, 19
SIN Function, 202
Single line if statement, 23
SIZE Statement, 203
SLIQ, 1, 5, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 21, 22, 23, 265
SOUND Statement, 204
Source Window, 266
SPACE Function, 205
SPC Function, 206
Special Symbols, 17
SPLITSCREEN Statement, 207
SQR Function, 208
STAMP Statement, 209
Statements, 23, 275
Statements not allowed here, 275
STATIC Statement, 210
Statistics, 5, 96, 177, 196, 227
Step Button, 266
Stop, 267
Stop Button, 267
STOP encountered, 280
STOP Statement, 211
Stopping a script, 16
STR Function, 212
String Constants, 18
STRING Function, 214
String not terminated, 269
String required after ALIAS, 278
String required after LIB, 278
STRING Type, 213
STRIPHIBIT Function, 215
SUB Statement, 216
Subroutine name expected, 271
Subroutines, 28, 50
Symbols, 17
Syntax error, 268
SYSTEM Statement, 217

T

TAB Function, 218
TAN Function, 219
Terminal, 181, 199
terminal window, 12, 16, 53, 257
THEN expected, 269
TIME Function, 220
TIMEOUT Function, 221
TIMER Function, 222
TO expected, 274
Tokens, 17
transfer protocol, 260
TRAPSCREEN Statement, 223
TXT, 195
TYPE ... END TYPE Statement, 224
Type character illegal for SUB, 271
Type character not allowed, 276
Type mismatch, 269
Type name expected, 272
Type specification illegal, 272
Types, 19, 20, 21

U

UCASE Function, 226
Undeclared identifier, 275
Unexpected end of file, 280
UNTIL or WHILE expected, 270
UPLOAD Function, 227
User defined type variable expected, 276
User Defined Types, 20
User ID, 259

V

VAL Function, 229
Variable expected, 271
Variables, 21, 47, 90, 136, 150, 213, 224, 248
VERSION Function, 230
VIEWFILE Statement, 231
VIEWGIF Statement, 232

## W

WAITFOR Statement, 233
Watch Button, 267
Watch Window, 266
WEND expected, 271
WEND not allowed here, 271
WEND Statement, 234, 240
WHEN ... GOTO executed from noncurrent scope, 282
WHEN CLEAR Statement, 235
WHEN condition expected, 276
WHEN DISABLE Statement, 236
WHEN ENABLE Statement, 237
When statement, 27, 238
WHILE ... WEND Statement, 240
WINVERSION Function, 241

## X

Xmodem, 96, 177
Xon/Xoff, 242
XONXOFF Statement, 242
XOR Operator, 243

## Z

Zmodem, 66, 97, 178, 197, 228, 260, 261

Mustang Software, Inc.
P.O. Box 2264, Bakersfield, CA 93303